

GENOLVE: A BIOINFORMATICS TOOL FOR THE STUDY OF STRUCTURE-DRIVEN GENOME EVOLUTION

by

Helene Fouche



*Thesis presented in fulfilment of the requirements for
the degree of Master of Science in the Faculty of Science
at Stellenbosch University*

Supervisor: Prof Hugh-George Patterson

Co-supervisor: Prof Johan Burger

March 2021

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2021

Copyright © 2021 Stellenbosch University

All rights reserved

Abstract

The process of structural genomic evolution, how one genome may have evolved into another via large-scale rearrangement events, has been a topic of study for the last few decades. Numerous genomic sorting models and algorithms that transform one genome into another using genome rearrangements exist. From a biological perspective, some of the models are more accurate than others. The rDCJ model is currently the best in this respect. In order to gain an in-depth understanding of the underlying mechanisms of genome evolution, knowledge of the different ways in which one genome may have evolved into another, is vital. At present, bioinformatic tools do exist that implement one or more of the genomic sorting models, however they output only one of an often-vast amount of equiprobable rearrangement scenarios.

This thesis describes amendments made to the existing rDCJ genomic sorting model, which increases its biological accuracy. The amended model was incorporated into an algorithm that finds all of the most parsimonious rearrangement scenarios for sorting one genome into another. This allowed for the development of the open-source, command-line, bioinformatics tool, Genolve.

When tested on simulated data, Genolve transformed one input genome into another 100% of the time, generating the full set of the most parsimonious sorting scenarios in its output. Unique to Genolve is also an output-summary of the average number and type of rearrangements present in the set of rearrangement scenarios. Application to two related strains of the yeast, *Saccharomyces cerevisiae*, illustrated the utility of the tool in analysing biological data.

Making Genolve available as an open-source tool will allow researchers interested in the process of genomic evolution to conduct comprehensive studies in greater depth than was previously possible.

Acknowledgements

I would like to thank my supervisor, Professor Hugh-George Patterton, for his guidance, remarks and useful comments during the learning process of this thesis.

Additionally, I would like to express my gratitude to the National Research Foundation for their monetary assistance with my postgraduate studies.

Table of Contents

Chapter 1: Literature Review	16
1.1. Introduction.....	16
1.1.1. The process of genome evolution	16
1.1.2. Structural alterations	16
1.2. The study of structural alterations.....	17
1.3. Genome rearrangement problems	19
1.4. Genomic Sorting models and algorithms	21
1.4.1. Sorting by inversions	21
1.4.2. Sorting by translocations.....	21
1.4.3. Sorting genomes with different chromosome ends	22
1.4.4. The transposition operation.....	23
1.4.5. The block-interchange operation	23
1.4.6. Cut and/or join models.....	24
1.5. Bioinformatic tools for the study of large-scale genomic evolution.....	36
Chapter 2: The development of Genolve	39
2.1. Brief description of the tool	39
2.2. Introduction: The rDCJ model and its limitations.....	39
2.2.1. The DCJ operation	39
2.2.2. The restricted DCJ model: an improvement on the DCJ model	40
2.2.3. Problems with the rDCJ model	40
2.3. Development of the tool.....	43
2.3.1. Notation and terminology	43
2.3.2. I: Initial coding of the rDCJ model	46
2.3.3. II: The single solution problem.....	50
2.3.4. III: The operation cost problem	58
2.3.5. IV: The block-interchange problem.....	62
2.3.6. V: From DCJ operations to genome rearrangements.....	72
2.3.7. VI: Incorporating the expected frequency of occurrence per rearrangement via a weighting system	72
2.3.8. The output of Genolve and how to interpret it.....	76
Chapter 3: Evaluation of Genolve using synthetic data	82
3.1 Development of synthetic gnomes and genomic complexity notation	83
3.1.1 Development of synthetic genomes	83

3.1.2 Genomic complexity notation	86
3.2 Genolve's capability to solve source-target genome input pairs.....	87
3.3. Genolve's performance under different weighing ratios	88
3.3 The size of solution sets and the length of solutions.....	95
3.3.1 The average number of solutions per solution set.....	95
3.3.2. The average length of a solution	105
3.6 The runtime and memory usage of Genolve.....	111
3.6.1 The runtime of Genolve	111
3.6.2 Space complexity and memory usage of Genolve.	120
Chapter 4: Testing Genolve on biological data.....	126
4.1. <i>Saccharomyces cerevisiae</i>	126
4.2. Mauve	127
4.3. The process	128
4.3.1 Generating the input files for Genolve.....	128
4.3.2. Running Genolve	136
4.3.3 Comparison with GRIMM and UniMoG.....	144
4.3.4. Analysis of sequence regions at the breakpoints identified by Mauve	156
Chapter 5: Conclusion and Future prospects	160
5.1. Conclusion	160
5.1.1. Computational and memory complexity of Genolve	162
5.2 Future prospects	163
5.2.1 Genome content altering events.....	164
5.2.2 Biological constraints.....	170
5.3. Final Thoughts	174
Reference List	176

List of Figures

Fig 1. Types of structural changes that can occur during the evolution of genomes. Sequence regions in the genome are represented by labeled block arrows, with the direction of the arrow head and the sign preceding the label indicating the orientation of the sequence region in the genome. a) An inversion event affecting blocks {2, 3}. b) An unbalanced translocation of block {1, 2}. c) A balanced translocation resulting in the exchange of blocks {1, 2} and {6, 7}. d) Chromosome fusion and fission, with the adjacency between blocks {2, 3} being broken and formed. e) A transposition of blocks {2, 3} to between blocks {6, 7}. f) An inverted transposition of blocks {2, 3} inserting blocks {-3, -2} between blocks {6, 7}. g) Insertion and deletion of blocks {4, 5}, the red crosses over these blocks in the bottom genome indicating their absence. h) Duplication of block {2}.18

Fig 2. Simple example of one permutation being sorted into another by application of a series of rearrangement operations. The initial permutation [1, 4, -6, -5, 7, 2, 3, 8] can be arranged (sorted) into the identity permutation [1, 2, 3, 4, 5, 6, 7, 8] by performing a transposition operation followed by an inversion. The subsets of the permutation that are affected by each rearrangement are underlined.20

Fig 3. Sorting by inversion. a) The sorting of the unsigned permutation [1, 4, 3, 2, 5] using inversions. b) The sorting of the signed permutation [1, 4, 3, 2, 5] using inversions. In each case the subset of the permutation that is inverted is underlined.22

Fig 4. A block-interchange operation. Sequence regions in the genome are represented by labeled block arrows, with the direction of the arrow head corresponds to the orientation of the sequence in the genome. The block-interchange operation shows the swapping.....24

Fig 5. Genome graph for the genome $G = \{(\circ 3, -2, -4, -6, 5, 1 \circ) (8, -7, -9, 10)\}$. The graph is composed of directional edges (the genes) connected to vertexes (the extremities). A linear and circular chromosome is shown.25

Fig 6. a) Breakpoint graph $BP(G_A, G_B)$ for the circular genomes (a) $G_A = \{(1, 2, 3, 4, 5, 6, 7)\}$ and $G_B = \{(1, -3, -2, 4) (5, 6, -7)\}$. (b) The adjacencies in G_A are represented by black edges and those in G_B by grey edges. (c) Breakpoint graph $BP(G_A, G_B)$ for the identical circular genomes $G_A = G_B = \{(1, 2, 3, 4, 5, 6, 7)\}$. The adjacencies in G_A are represented by black edges and those in G_B by grey edges. The identical adjacencies in the two genomes are seen as one-cycles (cycles of length two) in the graph.26

Fig 7. a) Adjacency graph $AG(G_A, G_B)$ for the circular genomes shown in Fig 6a. The adjacency graph consists of four cycles, two of which are one-cycles representing identical adjacencies between the genomes. b) Genomes G_C and G_D , composed of two linear chromosomes $G_C = \{(\circ 1, 2, 3, 4, 5 \circ) (\circ 6, 7, 8, 9 \circ)\}$ and two linear and one circular chromosome $G_D = \{(\circ 1, 4, -3, 2 \circ) (5, -6, 7) (\circ 8, -9 \circ)\}$, respectively. c) Adjacency graph $AG(G_C, G_D)$. d) The identical genomes G_E and G_F where $G_E = G_F = \{(\circ 1, 2, 3 \circ) (4, 5, 6)\}$. The matching adjacency graph $AG(G_E, G_F)$ consists of only one-cycles and paths of length one.28

Fig 8. a) The cut and join operation. A cut operation followed by a join operation is shown, applied to the genome $G_A = \{(\circ 1, 2, 3 \circ) (\circ 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 2 \circ) (\circ 3, 4 \circ)\}$. The position of the cut is shown by the vertical dashed line, and the join operation by the horizontal dashed line. b) The corresponding adjacency graph of the cut and of the join operation.29

Fig 9. a) Cut-and-join operation. A cut-and-join operation is applied to the genome $G_A = \{(\circ 1, 2, 3 \circ) (\circ 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 4 \circ) (\circ 3, 4 \circ)\}$. The cut component of the operation is represented by the vertical dashed line and the join component by the horizontal dashed line. b) The corresponding adjacency graphs.30

Fig 10. a) Double-cut-and-join operation applied to the genome $G_A = \{(\circ 1, -3, -2, 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 2, 3, 4 \circ)\}$. The cut components of the operation are represented by the vertical dashed lines and the join components by the horizontal stippled lines. b) The matching adjacency graph.32

Fig 11. Different ways in which the cut-and-join operation can act on vertices. a) The internal vertices $u = \{p, q\}$ and $v = \{r, s\}$ can be replaced by $\{p, r\}$ and $\{q, s\}$ resulting in an inversion, or by the vertices $\{p, s\}$ and $\{q, r\}$, resulting in an excision and circularization. b) The internal vertex $u = \{p, q\}$ and external vertex $v = \{r\}$ can be replaced by $\{p, r\}$ and $\{q\}$ resulting in an inversion or by the vertices $\{p\}$ and $\{q, r\}$ resulting in breaking off and circularization of an end of a chromosome. c) The external vertices $u = \{p\}$ and $v = \{r\}$ are replaced by the internal vertex $\{p, r\}$, resulting in a chromosome fusion. d) The internal vertex $\{p, r\}$ is replaced by the external vertices $\{p\}$ and $\{r\}$, resulting in a chromosome fission. u and v represent either an adjacency between two genes or a telomeric end. p, q, r and s represent the gene extremities making up the adjacencies and telomeric ends.33

Fig 12. Optimal sorting scenarios for sorting genome $G_B = \{(\circ 1, 4, 3, 6, 2, 5, 7 \circ)\}$ into genome $G_A = \{(\circ 1, 2, 3, 4, 5, 6, 7 \circ)\}$. The genomic distance between the genomes is 4 as both sorting scenarios comprise 4 steps. The red dashed lines show which adjacencies will be broken in that step. a) an optimal sorting scenario when sorting by general DCJ operations. Note the presence of multiple circular chromosomes during certain stages of the transformation process. b) an optimal sorting scenario when sorting by restricted DCJ operations.35

Fig 13. Sorting scenario generated by the GRIMM tool for the transformation of the source genome $[[1, 3, 2, 8], [-5, 6, 7, 4]]$ into the target genome $[[1, 2, 3, 4], [5, 6], [7, 8]]$. The colouring of the integers representing the sequence blocks correspond to the chromosomes on which they are located in the target genome. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.37

Fig 14. The sorting scenario generated by UniMoG for sorting the same source and target genomes as in Fig 13. The colouring under sequence blocks in one step correspond to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.38

Fig 15. Depiction of the sorting scenario for transforming the genome $[[1, 2, 3], [6, 7, 8, 4, 5, 9, 10]]$ into the genome $[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]$ using two different series of rearrangements a) Two successive unbalanced translocations are used for the transformation b) A single transposition, consisting of an excision and circularization step followed by a linearization and reinsertion step, is used for the transformation.42

Fig 16. Illustration of the how the Genolve algorithm recursively generates a dictionary containing the source, target and all intermediary genomes. In the figure the process occurs from left to right. With the left most path being completed before proceeding to the next left most path. Genomes already present in the dictionary are not added a second time; therefore, paths terminate at the second occurrence of Intermediates 3, 8 and 4 respectively.55

Fig 17. The network of solutions constructed from the genomes present in the dictionary created in Fig 16. The network shows different paths that can be taken to get from the source genome to the target genome i.e. to achieve the successful transformation of the source into the target genome.57

Fig 18. Illustration of the difference between transpositions and block-interchanges using the following source, target genome pair: source genome = $[[1, 4, 5, 2, 3, 6]]$ and target genome = $[[1, 2, 3, 4, 5, 6]]$. Arrows represent the genes with the integer label of a gene positioned under the arrow. Telomers and adjacencies are show at the edges and at the meeting point of two genes respectively. Red adjacencies represent a newly created adjacency that resulted in the circularization of a segment of sequence blocks. The red dashed lines are used to indicate the adjacency that will be destroyed in order to linearize the circular component. a) Shows a transposition operation used to achieve genome transformation. The adjacency formed during the circularization step is equivalent to the one that is destroyed during the proceeding linearization step. b) Transformation is achieved using a block-interchange event. The adjacency created during circularization differs from the one that is destroyed during the following linearization step.62

Fig 19. Illustration of the two transposition events that constitute a single block-interchange event. The block-interchange event is comprised of the swapping of sequence blocks 2 and 3 with sequence blocks 6 and 7. This event

involves the transposition of sequence blocks 6 and 7 to in between sequence block 5 and 2 (shown in green) and the transposition of sequence blocks 6, 7, 4 and 5 to in between sequence blocks 3 and 8 (shown in blue). 71

Fig 20. a) Depiction of how the transformation of the source genome $[[1,-3,2],[4],[6,5,7]]$ into the target genome $[[1,2,3],[4,5],[6,7]]$ can be divided into two independent parts involving sequence blocks 1, 2 and 3 and sequence blocks 4, 5, 6 and 7 respectively. Part one, the sorting of the chromosome $[1,-3,2]$ into $[1,2,3]$ can be achieved using either (i) one transposition event that compromises a circularization (T1a) followed directly by a linearization (T1b) operation or by using (ii) two inversions (I1 and I2). The second part achieving the transformation of chromosomes $[4],[6,5,7]$ into $[4,5],[6,7]$ is obtainable either by application of a transposition event constituting a circularization (T2a) and circularization (T2b) operation or by the use of two unbalanced translocations (U1 and U2). b) The network of solutions showing all possible combinations of the different rearrangement event that can be used to successfully transform the source into the target genome. There are a total of 14 different solution or paths through the network (shown at the bottom). c) a table showing different weighting ratios that can be applied to the network (left) and the path that would be include in the output of an algorithm finding the lowest cost/weight paths through the network (right)..... 77

Fig 21. An example of the output generated by the Genolve tool. The above output shows the results for the input genome pair: source genome = $[[10, 1, -2, 3, 5, -4, 9], [7], [8, 6], [11]]$ and target genome = $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]$. The '...' at the bottom of the figure indicates that only three of the 720 solutions generated by Genolve are shown in the figure. All solutions are present in the output file however. Each line within a solution contains (a genome, (the type of rearrangement that resulted in the genome, ((the adjacencies/telomeric ends that were destroyed/'cut' during the rearrangement), (the adjacencies/telomeric ends that were newly created by the rearrangement)), with x denoting the tail extremity and x.5 the head extremity of gene x. A genome is contained within a set of square brackets and each individual chromosome making up a genome is also contained within a set of square brackets. If the first index of a chromosome is 'o', it indicates the circular nature of that chromosome. The following abbreviations for the different type of rearrangements are used: inversion: inv, circularization operation of a transposition/block-interchange: trp0, reinsertion operation of a transposition: trp1, reinsertion operation of a block-interchange: trp2, balanced translocation: b_trl, unbalanced translocation: u_trl, chromosome fission: fis, chromosome fusion: fus. 78

Fig 22. Genome graphs giving a visual representation of the source, target and intermediary genomes present in the different lines of Solution 1 present of the solution set generated by Genolve for the source genome: $[[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], [11]]$ and target genome: $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]$. a) – h) shows the various genomes present in lines 1 – 8 of Solution 1 respectively. Each arrow represents a gene, with the gene label appearing above the arrow. Genes occur in the tail-to-head direction (the tail of the arrow precedes the arrow head) unless the orientation of the gene in the genome is inverted, in which case the head of the arrow representing the gene will precede the tail of the arrow. The inverse orientation of a gene within the genome is also indicated by the negative sign in front of a gene label on top of an arrow. The output generated by Genolve for this source-target genome pair is shown in Fig 21..... 81

Fig 23. a) Illustration of how the Evolver generates a source genome from a target genome by applying a series of rearrangements to it. In this example four rearrangement events are used for source genome generation. b) shows the inverse of the events in a) and describes the 'true' evolutionary scenario separating the source genome from the target genome..... 84

Fig 24. Illustration of how a source-target genome pair of which the source genome contains final-state adjacencies can be condensed. The adjacency (2.5, 3) in the target genome, between sequence blocks 2 and 3 (pink) is also present between sequence blocks -3 and -2 (pink). These two sequence blocks are condensed into a single sequence block in the bottom genomes. In purple at the top the adjacency (4.5, 5) exists between sequence blocks 4 and 5 in both the target and source genome. These sequence blocks are thus also condensed into a single sequence block in the genomes at the bottom. This condensing of genomes results in a reduction in the number of sequences blocks the genomes

consist of. At the top the uncondensed genomes consist of seven sequence blocks each. In contrast their condensed counter parts in the bottom of the figure consist of only five sequence block each. 87

Fig 25. The percentage time the 'true' evolutionary is present in the solution set generated by Genolve is plotted against increasing measures of genome complexity under a one-to-one weighting ratio (red), same-as-solution ratio (blue) and pseudo-randomized ratio (green). For each data point shown, the percentage time the 'true' scenario was present in the solution set is given for 100 000 runs. 90

Fig 26. a) Illustration of how the Evolver generates a source genome from a target genome by applying a series of rearrangements to it. Four rearrangement events are used for source genome generation, the last of which is the inversion of sequence block [-7]. b) shows the inverse of the events in a) and describes the 'true' evolutionary scenario separating the source genome from the target genome. The first rearrangement is the inversion of sequence block [7] (outlined in red). The inversion of [7] creates the adjacencies (3.5, 7.5) and (7, 9.5). Neither of these are final-state adjacencies and the execution of the DCJ operation resulting in the inversion of [-7] is therefore invalid. 92

Fig 27. The percentage time the 'true' evolutionary is present in the solution set generated by Genolve is plotted against increasing measure of genome complexity under a one-to-one weighting ratio (red), same-as-solution ratio (blue) and pseudo-randomized ratio (green). For each data point shown, the percentage time the 'true' scenario was present in the solution set is given for 100 runs. 94

Fig 28. The number of solutions within a solution set is plotted at increasing levels of genomic complexity. Each data points shows the average number of solutions within a solution set across 100 000 runs. The results under each of the different weighting ratios, one-to-one ratio (red), same-as solution ratio (blue) and pseudo-randomized ratio (green) is plotted on the graph. 97

Fig 29.1. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 3/4 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range ($Q_3 - Q_1$). The median of each data set is indicated by the orange line. 98

Fig 29.2. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 4/5 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range ($Q_3 - Q_1$). The median of each data set is indicated by the orange line. 988

Fig 29.3. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 5/6 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range ($Q_3 - Q_1$). The median of each data set is indicated by the orange line. 988

Fig 29.4. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 6/7 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range ($Q_3 - Q_1$). The median of each data set is indicated by the orange line. 989

Fig 29.5. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 7/8 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range (Q_3-Q_1). The median of each data set is indicated by the orange line. 98

Fig 29.6. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 8/9 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q_3) and first quartile (Q_1) respectively by 1.5 x the interquartile range (Q_3-Q_1). The median of each data set is indicated by the orange line. 98100

Fig 30. Output generated by the Genolve tool for the following input genome pair, source genome: $[[1, 2, 3, 4, 6, 5]]$, target genome: $[[1, 2, 3], [4, 5, 6]]$. The solution set contains six solutions, all of which consist of the same three independent rearrangements, an inversion, a chromosome fission and a transposition. The six solutions are the different permutations, $3!$, of these rearrangements. The ordering of the rearrangement for each solution is shown in red next to the solution number. 102

Fig 31. Output generated by the Genolve tool for the following input genome pair, source genome: $[[1, -3, -2, 4]]$, target genome: $[[1, 2], [3, 4]]$. The solution set contains two solutions, the first of which consists of a chromosome fission followed by a balanced translocation, and the second of an inversion followed by a chromosome fission. The symbols of the different rearrangement events are underlined in red. 104

Fig 32.1. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 3/4. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 32.2. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 4/5. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 32.3. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 5/6. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 32.4. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 6/7. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 32.5. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 7/8. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 32.6. scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 8/9. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value. 106

Fig 33. The generation of a source genome from a target genome. Three rearrangements are applied to the target genome. The effects of the first rearrangement, an inversion of sequence block [2] is reverse by application of the final rearrangement, an inversion of sequence block [-2]. 109

Fig 34. An illustration of how the same source-target genome pair can be solved using alternatively a) three successive inversion events or b) a single transposition event. 110

Fig 35. The average time required by Genolve to generate a solution set for a pair of input genomes is plotted at increasing levels of genomic complexity. Each data points shows the mean value across 100 000 runs. The results under each of the different weighting ratios, one-to-one ratio (red), same-as solution ratio (blue) and pseudo-randomized ratio (green) are plotted on the graph. 112

Fig 36.1. a box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 3/4 is shown. a) outliers are included in the plot b) outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile (Q3) and first quartile (Q1) respectively by 1.5 x the interquartile range (Q3-Q1). The median of each data set is indicated by the orange line. 114

Fig 37. A scatter plot of the time (seconds) Genolve takes to solve the same input genome pair for 1000 runs. The average time taken by Genolve = 0.08187 ± 0.00528 seconds. 117

Fig 38. An illustration of how the worst-case time complexity for the Genolve algorithm can be calculated. The symbol n represents the number of non-final-state adjacencies present in the source genome. The tree structure shows how the number of branches splitting from any one point in a path relates to n . The runtime up to a particular level of the network of solutions (going from the source genome down to the target genome) is show in red on the left. 119

Fig 39. the results summary and first solution with in the solution set for a) run 1 at the genomic complexity level of 16/17 and b) run 6 at the genomic complexity level of 16/17. Inv: inversion, trp0: circularization step of the transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion. 124

Fig 40. Alignment by the MauveAligner of the genomes of the strains FJSA40.2 (top) and MTZ13.12 (bottom). Locally colinear blocks are shown as coloured regions occurring in a different order in the two sequences. 128

Fig 41. a) permutation matrix generated by Mauve showing the order of the locally colinear blocks in the genome of strain FJSA40.2 (top) and MTZ13.12 (bottom). The negative sign in front of block 8 in the bottom row indicates the inversed orientation of the block relative to its homologous counterpart in the top row. b) the base pair locations of the locally colinear blocks for genomes FJSA40.2 (left-most two columns) and MTZ13.12 (right-most two columns), the negative signs in the eighth entries of the right-most two rows indicates the reverse orientation of the region. 129

Fig 42. Results summary and first solution within the solution set generated by Genolve for an input genome pair of strain FJSA40.2 (target genome) and MTZ13.12 (source genome) of the yeast specie *Saccharomyces cerevisiae*. Inv: inversion, trp0: circularization step of the transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion. 137

Fig 43. Illustration of how a source-target genome pair, in which the number of final-state adjacencies in the source genome is non-zero, can be condensed into genome with few sequence blocks by eliminating those sequence blocks that are already in their final states. The sequence blocks to be eliminated in the top genomes are crossed out in red. At the bottom of the figures the condensed genomes, with the necessary sequence blocks removed is shown. The remaining sequence blocks are renumbered so that the target genome consists of consecutive integers.137

Fig 44. Results summary and first solution within the solution set generated by Genolve for the condensed input genome pair of strain FJSA40.2 (target genome) and MTZ13.12 (source genome) of the yeast species *Saccharomyces cerevisiae*. Inv: inversion, trp0: circularization step of the transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion.....138

Fig 45. Sorting scenario generated by the GRIMM tool for the same input genome as in Fig 43 The colouring of the integers representing the sequence blocks correspond to the chromosomes on which they are located in the target genome. Sequence blocks that will be affected by a rearrangement events are highlighted in yellow.145

Fig 46. Sorting scenario generated by UniMoG, under the restricted DCJ model, for sorting the same source and target genomes as in Fig 43. The colouring under sequence blocks in one step correspond to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.146

Fig 47. Output of the GRIMM bioinformatics tool for the input genome pair, target genome: [[1, 2, 3, 4, 5]] and source genome: [[1, 2, 4, 3, 5]]. The sequence block [4] was transposed to in between sequence blocks [2] and [3]. The GRIMM tool, unable to perform transpositions, transforms the source genome into the target genome using three reversals or inversions. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.149

Fig 48. Output of the GRIMM bioinformatics tool for the input genome pair, target genome: [[1, 2, 3], [4, 5, 6]] and source genome: [[1, 3], [4, 5, 2, 6]]. The sequence block [2] was transposed to inbetween sequence blocks [5] and [6]. The GRIMM tool, unable to perform transpositions, transforms the source genome into the target genome using two translocation events. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.149

Fig 49. Sorting scenario generated by UniMoG, under the restricted DCJ model, for the input genome pair, target genome: [[1, 2, 3], [4, 5, 6]] and source genome: [[1, 3], [4, 5, 2, 6]]. The sequence block [2] was transposed to in between sequence blocks [5] and [6]. The colouring under sequence blocks in one step correspond to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.....150

Fig 50. Screenshots of the results generated by UniMoG for the input genome in Fig 43 under the a) rDCJ, b) DCJ and c) HP model. Note that the screenshots exclude the graphics of the rearrangement scenario generated under each model as the purpose of this figure is to illustrate that under all three models, UniMoG states the rearrangement scenario shown is one of roughly 380067187500 (underlined in red) sorting scenarios.....152

Fig 51. Position of the CAF4 gene within chromosome 11 of the *Saccharomyces cerevisiae* reference genome. A red marker shows the location of the breakpoint within the gene. (Figure obtained from the NCBI gene graphic viewer platform).157

Fig 52. Graphic summary generated by the BLAST tool for the alignment of the CAF4 gene (the query sequence shown in blue) to chromosome 11 of the MTZ13.12 *Saccharomyces cerevisiae* strain (shown in red). The latter portion of the gene maps to position 524777bp – 525085bp on the MTZ13.12 chromosome.158

Fig 53. Graphic summary generated by the BLAST tool for the alignment of the CAF4 gene (the query sequence shown in blue) to chromosome 10 of the MTZ13.12 *Saccharomyces cerevisiae* strain (shown in red). The front portion of the gene maps to position 280522bp – 282218bp on the MTZ13.12 chromosome.158

Fig 54. Illustration of three different approaches to dealing with duplications prior to genome sorting. In each example one or more copies of gene '4' in G_A is matched to one or more copies of the gene in G_B (with unmatched copies being removed), prior to the sorting of G_B into G_A . The removal of a gene copy is shown by a red cross over the gene. The letters in the subscripts of gene '4' show which copies in the two genomes have been matched. a) Example of the 'exemplar method', where one copy of each gene duplicate per genome is selected and matched to the duplicate in the other genome. The rest of the copies are removed. b) Example of the 'intermediate strategy' where a number of copies of a duplicative gene are selected in one genome and matches to copies in a second genome. The unmatched copies of the gene in both genomes are removed. c) Example of the 'maximum matching strategy', where the maximum number of duplicates of a gene are selected in one genome and matched to gene copies in a second genome, while the unmatched copies are removed.165

Fig 55. Adjacency graphs for the genomes of which the genic content is not equal, $G_A = \{(\circ, 1, 2, 3, 4, \circ)\}$ is represented by black vertices, and $G_B = \{(\circ, 1, 4, \circ)\}$ by grey vertices. 'Ghost' vertices are used to represent adjacencies between gene extremities not present in G_B . a) ghost vertices are used to form adjacencies $\{2_h 3_i\}$ and $\{3_h 2_i\}$. This results in an adjacency graph consisting of two paths of length one, a cycle and a one-cycle containing a ghost vertex. b) ghost vertices are used to form different adjacencies, $\{3_i 2_i\}$ and $\{2_h 3_h\}$, thereby avoiding the formation of a one-cycle containing a ghost vertex.166

Fig 56. Adjacency graphs for the genomes of which the genic content is not equal, $G_A = \{(\circ, 1, 2, 3, 4, \circ)\}$ is represented by black vertices, and $G_B = \{(\circ, 1, 4, \circ)\}$ by grey vertices. 'Ghost' vertices are used to represent adjacencies between gene extremities not present in G_B . a) ghost vertices are used to form adjacencies $\{2_h 3_t\}$ and $\{3_h 2_t\}$. This results in an adjacency graph consisting of two paths of length one, a cycle and a one-cycle containing a ghost vertex. b) ghost vertices are used to form different adjacencies, $\{3_t 2_t\}$ and $\{2_h 3_h\}$, thereby avoiding the formation of a one-cycle containing a ghost vertex.168

Fig 57. a) A 2D representation of a potential 3D configuration for the genome $G_A = \{(\circ 4, 2, 8, 6 \circ) (\circ 1, -5, -3 \circ) (\circ -9, -7 \circ)\}$. The numbers represent those genes making up a chromosome. The spatial proximity of adjacencies between genes are highlighted by the grey dashed circles. Groups of adjacencies in close proximity are labeled by the alphabetical letters {a, b, c, d, e, f}. b and c) Illustrations of two different ways in which the adjacencies located in group b could be broken and reformed.173

List of Tables

<i>Table 1. Shows the number of runs conducted at various level of genomic complexity and the percentage time Genolve was successful in transforming the source genome into the target genome.</i>	<i>88</i>
<i>Table 2. The number of solutions present within the solution set generated by Genolve for input genome pairs separated by an increasing number of independent rearrangements.</i>	<i>103</i>
<i>Table 3. The memory usage (kb) and run time (h:m:s) for each of the ten runs at each of the five different levels of genomic complexity.</i>	<i>121</i>
<i>Table 4. lengths of the respective chromosomes that make up the nuclear genome of the Saccharomyces cerevisiae strain FJSA40.2.</i>	<i>130</i>
<i>Table 5. lengths of the respective chromosomes that make up the nuclear genome of the Saccharomyces cerevisiae strain MTZ13.12.</i>	<i>130</i>
<i>Table 6. Start and end positions of the locally colinear blocks (LCBs) identified by the MauveAligner for strain FJSA40.2. The length of each of the LCBs as well as the colour of the LCB in the graphic generated by Mauve is shown in the right-most columns of the table.</i>	<i>131</i>
<i>Table 7. Start and end positions of the locally colinear blocks (LCBs) identified by the MauveAligner for strain MTZ13.12. The length of each of the LCBs as well as the colour of the LCB in the graphic generated by Mauve is shown in the right-most columns of the table.</i>	<i>132</i>
<i>Table 8. Chromosomes spanned by each of the locally colinear blocks (LCBs) identified by the MauveAligner in the genome of S. cerevisiae strain FJSA40.2. The LCBs are identifiable by the colours in which they appear on the graphic of the alignment generated by Mauve.</i>	<i>133</i>
<i>Table 9. Chromosomes spanned by each of the locally colinear blocks (LCBs) identified by the MauveAligner in the genome of S. cerevisiae strain MTZ13.12. The LCBs are identifiable by the colours in which they appear on the graphic of the alignment generated by Mauve.</i>	<i>133</i>
<i>Table 10. The locally colinear blocks identified by the MauveAligner for contained on each of the chromosomes of the genome of the S. cerevisiae strain FJSA40.2. The colour in column two correspond to the integer labels assigned to them in column 3.</i>	<i>134</i>
<i>Table 11. The locally colinear blocks identified by the MauveAligner for contained on each of the chromosomes of the genome of the S. cerevisiae strain MTZ13.12. The colour in column two correspond to the intergenic labels in column 3.</i>	<i>135</i>
<i>Table 12. Calculation of the total number of balanced and unbalanced translocations present across all solutions found by Genolve.</i>	<i>140</i>
<i>Table 13. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 1.</i>	<i>141</i>
<i>Table 14. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 2.</i>	<i>142</i>

<i>Table 15. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 3.</i>	143
<i>Table 16. Comparison between the output of GRIMM, UniMoG and Genolve generated for the input genome pair of S. cerevisiae strains FJSA40.2 and MTZ13.12 in Fig 43.</i>	146
<i>Table 17. The number of solutions generated from each of the rearrangement sets as well as the total number of solutions generates under the HP- rDCJ-, DCJ- and amended rDCJ- model.....</i>	155

Chapter 1: Literature Review

1.1. Introduction

1.1.1. *The process of genome evolution*

Genomes of all organisms change over time, allowing the selection of individuals that are better adapted to a changed set of conditions in a background of a continually changing environment. These genome changes include localized alterations in DNA sequence, including single nucleotide polymorphisms (SNPs), small insertions and deletions (indels) of only a few nucleotides, as well as larger, structural modifications. Structural alteration may involve fragments of hundreds of kilobases that translocate to different positions or invert orientations. It may also involve the loss of genetic material from the genome, or the addition of foreign DNA fragments to a chromosome. Although both small and large sequence changes are important in evolution, the larger changes are often associated with more significant evolutionary transitions, including speciation [1], and are of specific interest when considering the evolutionary history of a genome.

1.1.2. *Structural alterations*

Structural changes can be divided into two categories: (i) genomic *rearrangement* events, which alters the order and orientation of sequence regions in the genome and (ii) genome *content-altering* events, such as large insertions, deletions and duplications, which alters the amount of genetic material in the genome.

Advancing our knowledge of the mechanisms and requirements pertaining to structural variation would contribute to our overall understanding of the evolutionary process [2]. In addition to a more in-depth insight into the species' survival and adaptation, a better grasp of evolutionary mechanisms will also have a significant impact in fields where structural changes are known to be important, such as cancer research and the study of genomic disorders [3–6].

Events that fall under the category of *genomic rearrangements*, include inversions, translocations, chromosome fusions and fissions, as well as transpositions. Fig 1a-f illustrates the different types of genomic rearrangements that can occur. Directional 'blocks' of DNA that make up the genome are represented by block arrows, where the direction of the arrow and the sign of its label indicates

its orientation in the genome. An inversion event involves the reversal of a segment of DNA on a chromosome (Fig 1a). There are two types of translocation events that can occur: the relocation of a segment of DNA at a chromosome end to the end of another chromosome (unbalanced translocation) (Fig 1b), or the exchange of DNA regions located at the ends of two chromosomes (balanced translocation) (Fig 1c). Chromosome fission and fusion describe the disruption of a chromosome into two separate chromosomes, and the joining of two chromosomes to form a single chromosome, respectively (Fig 1d). Two types of transpositions have been reported in literature [7–10]. The first simply involves removing a segment of sequence and its insertion elsewhere in the genome (Fig 1e). The second constitutes removing a segment of sequence, with its inversion prior to being reinserted elsewhere in the genome (Fig 1f). Transpositions can occur both intra- and inter-chromosomally.

Genomic content altering events include insertions, deletions and duplications, which involve large segments of DNA. Insertions describe the addition of DNA sequence to a genome, whilst deletions involve removing a stretch of DNA (Fig 1g). Duplications constitute the addition of multiple copies of a sequence region to the genome (Fig 1h).

1.2. The study of structural alterations

The pioneering studies by Sturtevant in 1917 [11] uncovered substantial evidence to support the occurrence of genome rearrangements in the model organism, *Drosophila melanogaster*, by revealing that *Drosophila* strains, which originated from identical or distinct geographical locations, differed in their gene orientation. There have since been numerous other studies that illustrate the role of genome rearrangements in the molecular evolution of both prokaryotes and uni- and multi-cellular eukaryotes [12–16].

The suggestion that the extent of disorder of one genome compared to another can serve as a measure of *evolutionary relatedness*, was first made by Dobzhansky and Sturtevant [17,18], in response to their finding that a number of inversions of the genes located on chromosome 3 differed between *Drosophila miranda* and *Drosophila pseudoobscura*. Consequently, the problem of finding the minimum number of inversions required to transform one genome into another, was formulated by Sturtevant and Novitski [19].

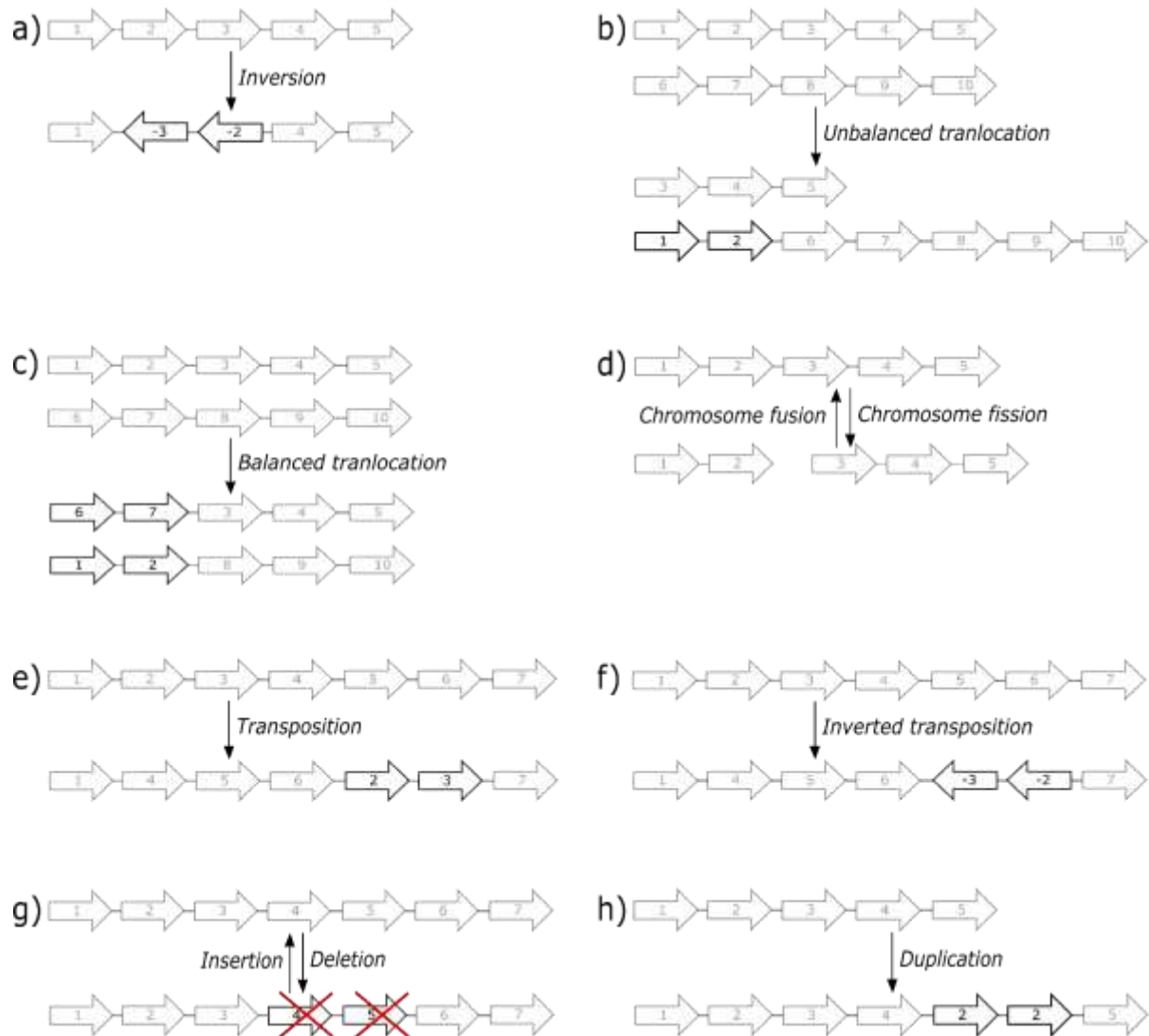


Fig 1. Types of structural changes that can occur during the evolution of genomes. Sequence regions in the genome are represented by labeled block arrows, with the direction of the arrowhead and the sign preceding the label indicating the orientation of the sequence region in the genome. a) An inversion event affecting blocks {2, 3}. b) An unbalanced translocation of block {1, 2}. c) A balanced translocation resulting in the exchange of blocks {1, 2} and {6, 7}. d) Chromosome fusion and fission, with the adjacency between blocks {2, 3} being broken and formed. e) A transposition of blocks {2, 3} to between blocks {6, 7}. f) An inverted transposition of blocks {2, 3} inserting blocks {-3, -2} between blocks {6, 7}. g) Insertion and deletion of blocks {4, 5}, the red crosses over these blocks in the bottom genome indicating their absence. h) Duplication of block {2}.

Genome rearrangement events occur at a low frequency, making it a reasonable assumption that the transformation of one genome into another will occur with the smallest number of

rearrangement events possible in a biological context [20,21]. Transformation scenarios that minimize the number of rearrangements are termed the most parsimonious scenarios, and it is the principle of parsimony that allows the problem of transforming one genome into another to be viewed and studied as a problem of optimizing a combination of events [21].

The study of the most parsimonious rearrangement scenarios between pairs of genomes was pioneered by Palmer and Herbon [22] in the late 1980s. They used the relative gene order of mitochondrial genomes of *Brassica oleracea* (cabbage) and *Brassica campestris* (turnip) to study the evolution of plant organelle genomes.

In 1982 Watterson *et al.* [23] suggested that the relative positions of genes in different circular genomes be represented as circular permutations, i.e. the genes are represented by integers or alphabetical letters arranged around a circle. Formulation of one possible evolutionary path between two genomes would then involve solving the problem of transforming the order of number or letters of one permutation into the other using successive inversion operations.

1.3. Genome rearrangement problems

Numerous genome rearrangement problems are currently being studied, including the genome halving problem, the genome median problem, the genomic distance problem and the genomic sorting problem. The *genomic sorting problem* – the problem of finding an optimal (most parsimonious) series of rearrangement operations with which one genome can be transformed into another – will be the main focus of this review. Finding an optimal series of rearrangements takes us one step closer to understanding what evolutionary events may separate two genomes. Where relevant, mention will also be made of the *genomic distance problem*, which concerns calculating the minimum number of operations required to transform one genome into another, without the need to compute each operation discretely. Genomic distance serves as an estimate of evolutionary relatedness between two genomes.

Fig 2 shows a simple example of how one genome, represented as a signed permutation (with negative signs indicating the inverse orientation of a sequence region in one genome relative to the other), may be sorted into another using a series of rearrangement operations.

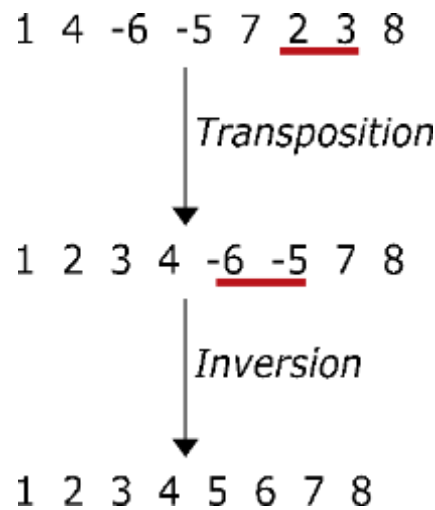


Fig 2. A simple example of one permutation being sorted into another by application of a series of rearrangement operations. The initial permutation $[1, 4, -6, -5, 7, 2, 3, 8]$ can be arranged (sorted) into the identity permutation $[1, 2, 3, 4, 5, 6, 7, 8]$ by performing a transposition operation followed by an inversion. The subsets of the permutation that are affected by each rearrangement are underlined.

Essentially, the genomic ordering or sorting problem can be described as a mathematical and computational challenge that has been studied for over two decades [24]. A comprehensive review of the earlier algorithms addressing this problem can be found in [25].

However, viewing the genomic sorting problem simply as a mathematical conundrum limits an opportunity for a biological insight, since a biological comprehension is required to ensure that excessively rare or unfeasible sorting operations are avoided. To this end, this review is centered on algorithms and models addressing the genomic sorting problem, with particular attention given to their *biological applicability and their accuracy*.

The review consists of three sections, followed by a brief conclusion. In Section 1, we discuss algorithms limited to sorting genomes with equal genic content, i.e. genomes affected only by genome rearrangement and not by content-altering events. The second section will focus on models and algorithms that have been extended to include content-altering operations, and the final section will give a brief review of genome sorting models and algorithms that incorporate biological constraints.

1.4. Genomic Sorting models and algorithms

1.4.1. Sorting by inversions

The first algorithm addressing the genomic sorting problem was an approximation algorithm [24], sorting *unsigned* permutations by successive inversion operations. By representing genomes as unsigned permutations, the orientation of sequence regions within the genome is inconsequential (Fig 3a). However, the orientation of a DNA sequence is highly relevant from a biological perspective, as it may place the promoter area of a gene adjacent to transcriptionally repressive heterochromatin, or, conversely, within a transcriptionally competent euchromatic environment.

A signed implementation of this algorithm was subsequently introduced [26], increasing its biological relevance. Numerous algorithms with improved runtime followed [27–31]. The most time-efficient algorithm for sorting by inversions is currently the sub-quadratic algorithm of Tannier and Sagot [32,33].

Fig 3 shows an example of sorting the permutation [1, 4, 3, 2, 5] into its identity permutation [1, 2, 3, 4, 5] by inversion operations. Fig 3a shows a sorting scenario for the unsigned version of the algorithm and Fig 3b the signed version. In the unsigned version, the permutation can be sorted by applying a single inversion because the orientation of each of the individual sequence regions is irrelevant. If the same inversion operation were to be applied to the signed version of the original permutation, the result would be [1, -2, -3, -4, 5] which is not equivalent to the sought identity permutation [1, 2, 3, 4, 5].

1.4.2. Sorting by translocations

Inversion occurs at a high frequency in genome evolution. A comparison of the *Saccharomyces cerevisiae* and *Candida albicans* genomes show more than a 1000 inversions since evolutionary divergence [34]. Despite its high prevalence, other types of rearrangements also occur. One such prominent rearrangement is translocation. Hannenhalli developed a polynomial-time algorithm for sorting multi-chromosomal genomes by translocation operations [35]. Bergeron *et al.* showed that Hannenhalli's algorithm was based on an incorrect assumption (see [36]), and subsequently corrected the algorithm [36].

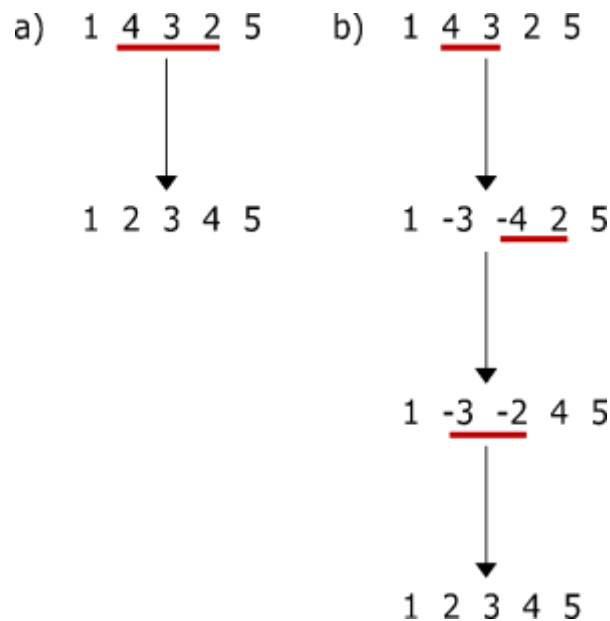


Fig 3. Sorting by inversion. a) The sorting of the unsigned permutation $[1, 4, 3, 2, 5]$ using inversions. b) The sorting of the signed permutation $[1, 4, 3, 2, 5]$ using inversions. In each case, the subset of the permutation that is inverted is underlined.

The similarity between the problems of sorting by translocation and sorting by inversion enabled the development of a sub-quadratic time algorithm for sorting by translocation [37], based on the algorithm of Tannier and Sagot [32].

These algorithms are all based on the basic assumption that the ends of the chromosomes, i.e. those genes adjacent to telomeres, are identical between the two genomes, meaning that do not account for chromosome fusion, fission nor unbalanced translocations.

1.4.3. Sorting genomes with different chromosome ends

Hannenhalli and Pevzner were the first to address instances of genomes with different chromosome ends [38]. Their model, called the *general HP model*, is capable of sorting linear, multi-chromosomal genomes with equal genic content via inversion and translocation, as well as chromosome fusion and fission. Since publication of the model, numerous researchers have identified problems with the algorithm (related to the optimal capping of chromosomes) and

several improved versions have been published [39–41]. The capping of chromosomes comprises the addition of distinct symbols, representing telomeres, to the ends of linear chromosomes.

1.4.4. The transposition operation

The final biological rearrangement event, is the transposition operation. The problem of sorting by transpositions is significantly more complex than that of sorting by inversions (see [42]). As with the initial sorting algorithms, the problem of sorting by transposition was found to be NP-hard [42], but there exist numerous approximation algorithms for solving it (an algorithm is termed a *k*-approximation algorithm if the solution it produces is guaranteed to be no greater than *k* times the optimal solution [43]). It is worth mentioning that the majority of sorting algorithms that incorporate transpositions account for only one type of this rearrangement, where the orientation of the transposed sequence region remains conserved (Fig 1e).

Bafna and Pevzner [44] were the first to investigate the problem of sorting by transposition, and their study produced a 1.5-approximation algorithm. There have since been various attempts to improve the approximation factor [45–47], the best of which is the 1.357-approximation algorithm developed by Elias and Hartman [47]. Various approximation algorithms for sorting by inversions and transpositions have also been presented over the last two decades [48–50], but as with the problem of sorting by transposition, the problem of sorting by transposition and is NP-hard [42,51].

1.4.5. The block-interchange operation

A generalized version of the problem of sorting by transposition, namely sorting by block-interchange, was introduced in [52]. A block-interchange operation entails exchanging two non-intersecting sequence regions or blocks of any length (Fig 4). If the sequence blocks that are swapped, are adjacent, the operation becomes a transposition operation (Fig 1e). Improvements to this algorithm were introduced by [7,53].

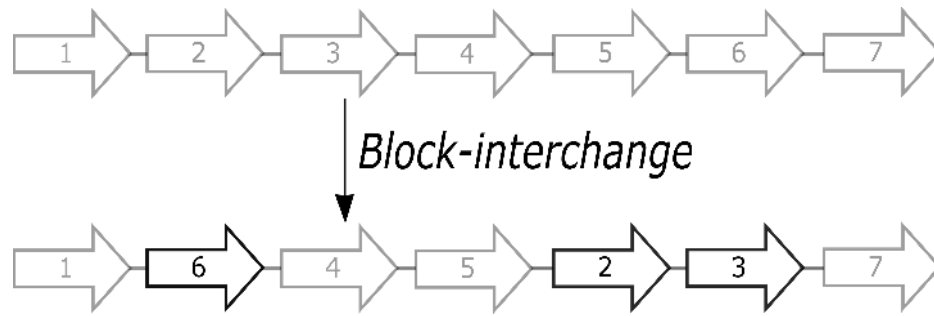


Fig 4. A block-interchange operation. Sequence regions in the genome are represented by labeled block arrows, with the direction of the arrowhead corresponds to the orientation of the sequence in the genome. The block-interchange operation shows the swapping

1.4.6. Cut and/or join models

More recent rearrangement models are the so-called cut and/or join models. (This review discusses these models in terms of the genomic sorting problem. For a comprehensive review of cut and/or join models, covering various other problems relating to genomic rearrangements, see [54]).

The cut and/or join models include the (i) single-cut *or* join model, (ii) single-cut *and* join model and (iii) double-cut and join model.

Before discussing the various models, defining a basic notation and graph structures used in the remainder of this section is useful.

1.4.6.1. Notation

A *gene* is a segment of DNA with a tail (x_t), followed by a head (x_h), called gene extremities (for inverted genes x_h precedes x_t). A *genome* composed of a set of genes is a set of adjacent gene extremities, where each gene extremity borders on one adjacency (region between genes), i.e. the end of one gene can be adjacent to at most the end of one other gene. A gene extremity can be replaced by a *telomere marker* (white circle), which signifies the end of a linear chromosome. Note that a gene is any sequence block in reality, but considering genes here simplifies the discussion.

The set G below, is a genome for the gene set $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$:

$$G = \{\{\circ, 3_t\}, \{3_h, 2_h\}, \{2_t, 4_h\}, \{4_t, 6_h\}, \{6_t, 5_t\}, \{5_h, 1_t\}, \{1_h, \circ\}, \{8_h, 7_h\}, \{7_t, 9_h\}, \{9_t, 10_t\}, \{10_h, 8_t\}\},$$

where each element $\{a, b\}$ is an adjacency between two genes, except when $a=\circ$ or $b=\circ$, when a telomeric adjacency is denoted.

A simpler notation for the above, and one that is used in the sub-section below, is:

$$G = \{(\circ 3, -2, -4, -6, 5, 1 \circ), (8, -7, -9, 10)\},$$

where the elements in round brackets represent a set of genes on a chromosome. If the first and last elements in a chromosome are telomeric symbols, \circ , then the chromosome being represented is linear; if not, the chromosome is circular.

This set G can be represented as a *genome graph* (Fig 5), where vertices (or nodes) are adjacencies between gene extremities and the edges are the genes. Genes are read in the tail-to-head direction as indicated by the arrowheads on the edges. Each vertex has a degree of either one or two, i.e. it has either one or two edges incident on it. Vertices of degree one signify the end of a linear chromosome. The set G represents a genome containing one linear chromosome, represented by a directional acyclic graph (DAG), and one circular chromosome. Note that the circular chromosome consists of 4 genes. By convention, the circle is read in a clock-wise fashion to assign the direction of each gene.

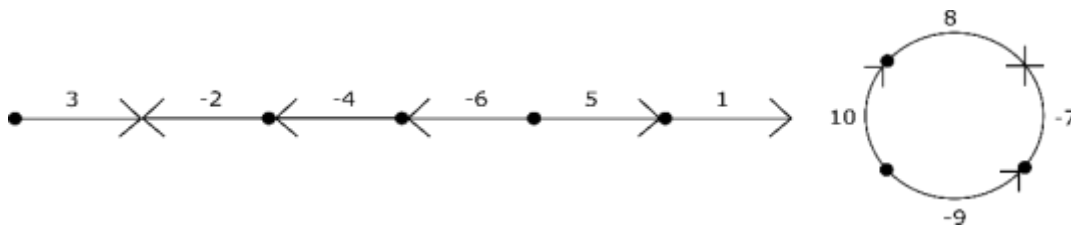


Fig 5. Genome graph for the genome $G = \{(\circ 3, -2, -4, -6, 5, 1 \circ) (8, -7, -9, 10)\}$. The graph is composed of directional edges (the genes) connected to vertexes (the extremities). A linear and circular chromosome is shown.

1.4.6.2. Breakpoint graph

The breakpoint graph was introduced by Bafna and Pevzner [26]. For two genomes, G_A and G_B , the *breakpoint graph* is defined as $BP(G_A, G_B) = G(V, E_A, E_B)$, where V are the vertices, that represent the gene extremities. E_A is the edges, representing the adjacencies in G_A (*black edges* in Fig 6) and E_B , those in G_B (*grey edges* in Fig 6). The gene composition, order, and orientation of genome G_A and G_B are shown in Fig 6a. $G_A = \{(1,2,3,4,5,6,7)\}$ and $G_B = \{(1,-3,-2,4) (5,6,-7)\}$. The breakpoint graph of genome G_A is represented by the vertices and black edges in Fig 6b. Genome G_B , derived from genome G_A rearranged into two chromosomes, is represented by the same vertices and by the grey edges in Fig 6b. A genome structure is deduced from a breakpoint graph by following consecutive sequence blocks (represented by adjacent vertices) to its neighbor via the linked adjacency edge back to the starting vertex. Note that separate chromosomes such as in genome G_B are not directly obvious from the connectivity of vertices in a breakpoint graph. Identical adjacencies in the two genomes are seen as cycles of length two, called *one-cycles*, in the breakpoint graph. If two circular genomes are identical, the breakpoint graph consists of only one-cycles (Fig 6c).

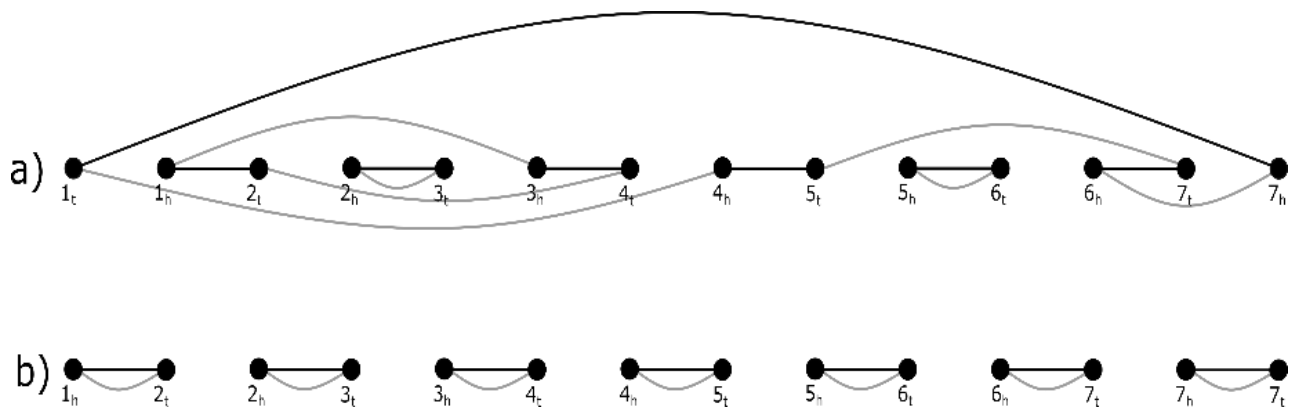


Fig 6. a) Breakpoint graph $BP(G_A, G_B)$ for the circular genomes (a) $G_A = \{(1, 2, 3, 4, 5, 6, 7)\}$ and $G_B = \{(1, -3, -2, 4) (5, 6, -7)\}$. (b) The adjacencies in G_A are represented by black edges and those in G_B by grey edges. (c) Breakpoint graph $BP(G_A, G_B)$ for the identical circular genomes $G_A = G_B = \{(1, 2, 3, 4, 5, 6, 7)\}$. The adjacencies in G_A are represented by black edges and those in G_B by grey edges. The identical adjacencies in the two genomes are seen as one-cycles (cycles of length two) in the graph.

1.4.6.3. Adjacency graph

The *adjacency graph* $AG(G_A, G_B)$, introduced by [55], is derived from the breakpoint graph and attempts to provide a simple visual representation of changes in the gene order and orientation between two genomes. As opposed to Fig 6, where gene adjacencies are represented by edges, adjacencies are collapsed into vertexes in the adjacency graph (see Fig 7). The gene order in a genome is read from the order of adjacent, labeled vertices in a row. The position of each gene extremity that defines the adjacency in a vertex is indicated in each of the two rows by an edge connecting identical extremities in a vertex. This gives an immediate visual impression of whether the gene position, orientation or connectivity has changed between genome G_A and G_B . The adjacency graph is shown in Fig 7a for the genomes G_A and G_B , which is defined in Fig 6a. Fig 7a. shows the adjacency graph for the circular genomes in Fig 6a. Fig 7b shows the linear genomes $G_C = \{(\circ 1, 2, 3, 4, 5 \circ) (\circ 6, 7, 8, 9 \circ)\}$ and $G_D = \{(\circ 1, 4, -3, 2 \circ) (5, -6, 7) (\circ 8, -9 \circ)\}$ with the corresponding adjacency graph shown in Fig 7c. The identical genomes G_E and G_F are shown in Fig 7d and the matching adjacency graph in Fig 7e.

The adjacency graph consists of a combination of paths which start and end at telomeres (terminal vertices with only one incident edge) and cycles that include only internal vertices. As with breakpoint graphs, shared adjacencies result in cycles of length two, referred to as one-cycles. If two genomes are identical, the adjacency graph consists only of one-cycles and paths of length one (Fig 7c). The construction of both breakpoint and adjacency graphs has a linear time and space complexity [55].

1.4.6.4. Single-cut or join model

Feijão and Meidanis [56,57] introduced the *single-cut or join (SCOJ)* model. The model employs adjacency graphs to solve the *genomic sorting problem* and involves two simple operations applied to the adjacencies and telomeres of a genome. A *cut* operation cuts the adjacency between two gene extremities and in so doing, creates two new telomeres (Fig 8a). A *join* operation is simply the reverse of the cutting process whereby two telomeres are linked together to form an adjacency between the gene extremities (Fig 8a). These two distinct operations constitute a *cut-or-join* operation. Fig 8a shows the transformation of the genome $G_A = \{(\circ 1, 2, 3 \circ) (\circ 4 \circ)\}$ into $G_B =$

$\{(\circ 1, 2, \circ) (\circ 3, 4 \circ)\}$ by the consecutive application of a cut and join operation. An adjacency graph for each of the two steps of the transformation is shown in Fig 8b.

A linear time algorithm exists for solving the genomic sorting problem under this model [56].

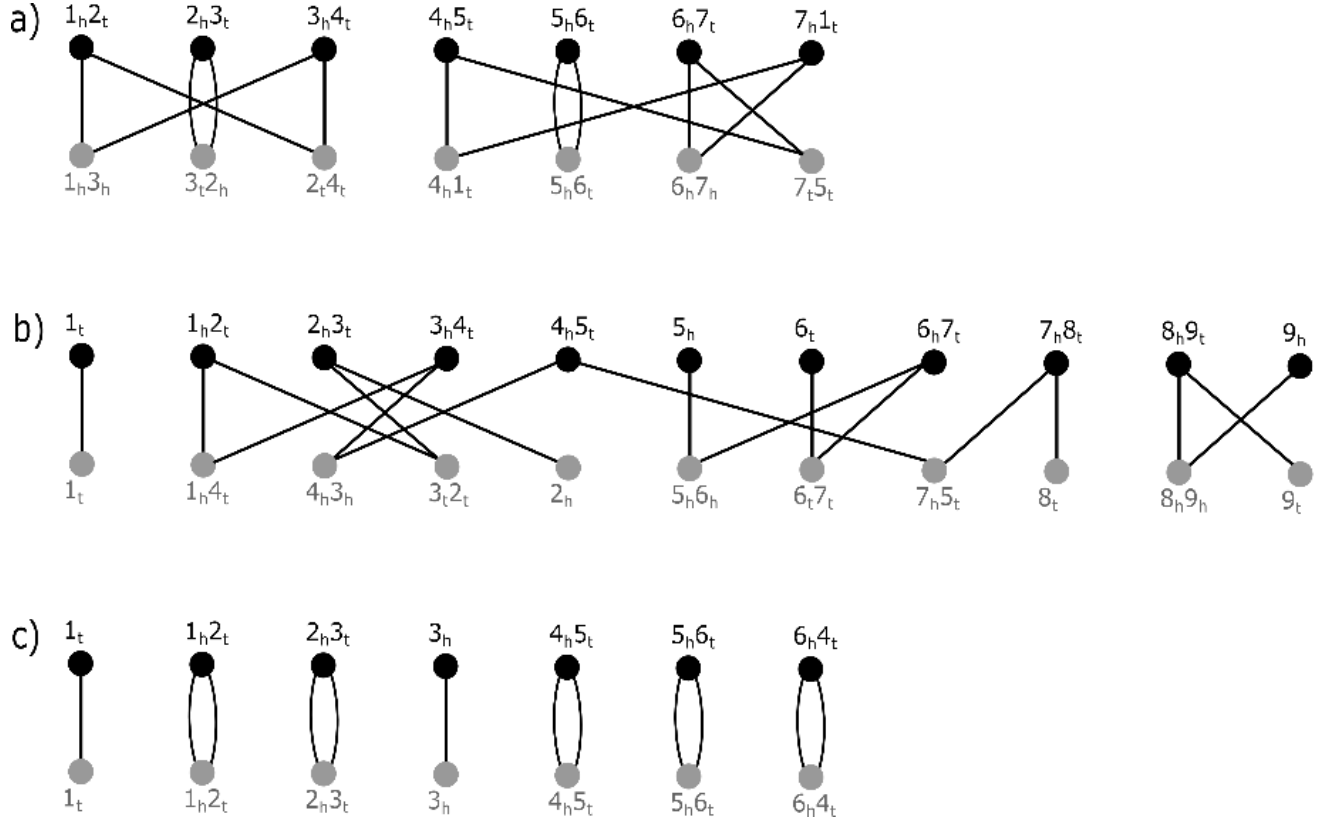


Fig 7. a) Adjacency graph $AG(G_A, G_B)$ for the circular genomes shown in Fig 6a. The adjacency graph consists of four cycles, two of which are one-cycles representing identical adjacencies between the genomes. b) Genomes G_C and G_D , composed of two linear chromosomes $G_C = \{(\circ 1, 2, 3, 4, 5 \circ) (\circ 6, 7, 8, 9 \circ)\}$ and two linear and one circular chromosome $G_D = \{(\circ 1, 4, -3, 2 \circ) (5, -6, 7) (\circ 8, -9 \circ)\}$, respectively. c) Adjacency graph $AG(G_C, G_D)$. d) The identical genomes G_E and G_F where $G_E = G_F = \{(\circ 1, 2, 3 \circ) (4, 5, 6)\}$. The matching adjacency graph $AG(G_E, G_F)$ consists of only one-cycles and paths of length one.

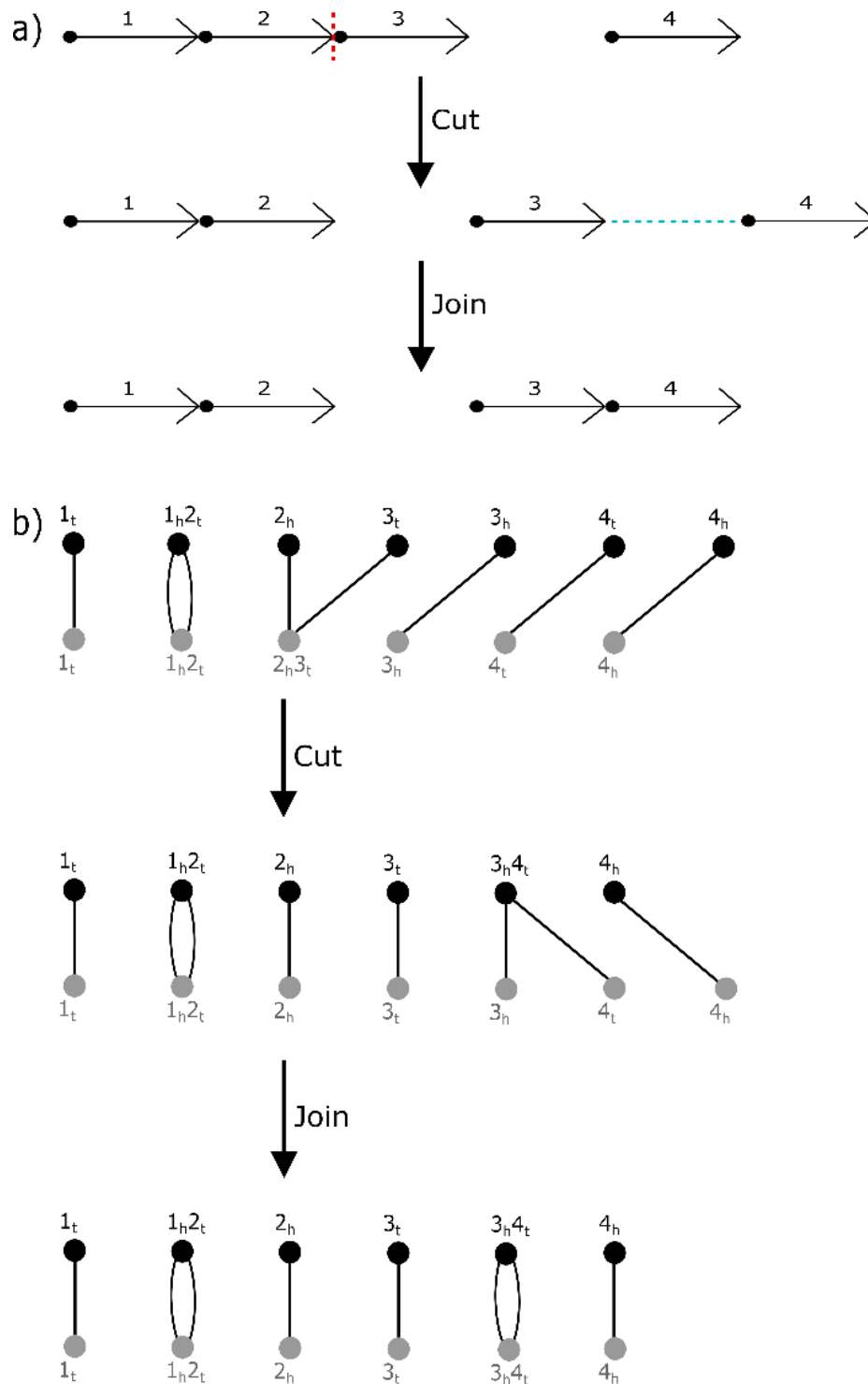


Fig 8. a) The cut and join operation. A cut operation followed by a join operation is shown, applied to the genome $G_A = \{(\circ 1, 2, 3 \circ) (\circ 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 2 \circ) (\circ 3, 4 \circ)\}$. The vertical dashed line shows the position of the cut and the horizontal dashed line the position of the join. b) The corresponding adjacency graph of the cut and the join operation.

1.4.6.5. Single-cut and join model

The *single-cut and join* model (SCAJ) [58], which introduces a cut and a join in a single operation, was introduced after the SCOJ model. The former differs from the latter in that the SCAJ model allows the additional *cut-and-join* operation. This operation entails breaking one adjacency and forming a new one between (potentially different) telomeres in a *single step* (Fig 9), instead of two distinct steps – the breaking of one adjacency followed by, step two, the forming of a new adjacency. This additional operation enables the model to account directly, not only for chromosome fission and fusion, but the inversion of chromosome ends as well.

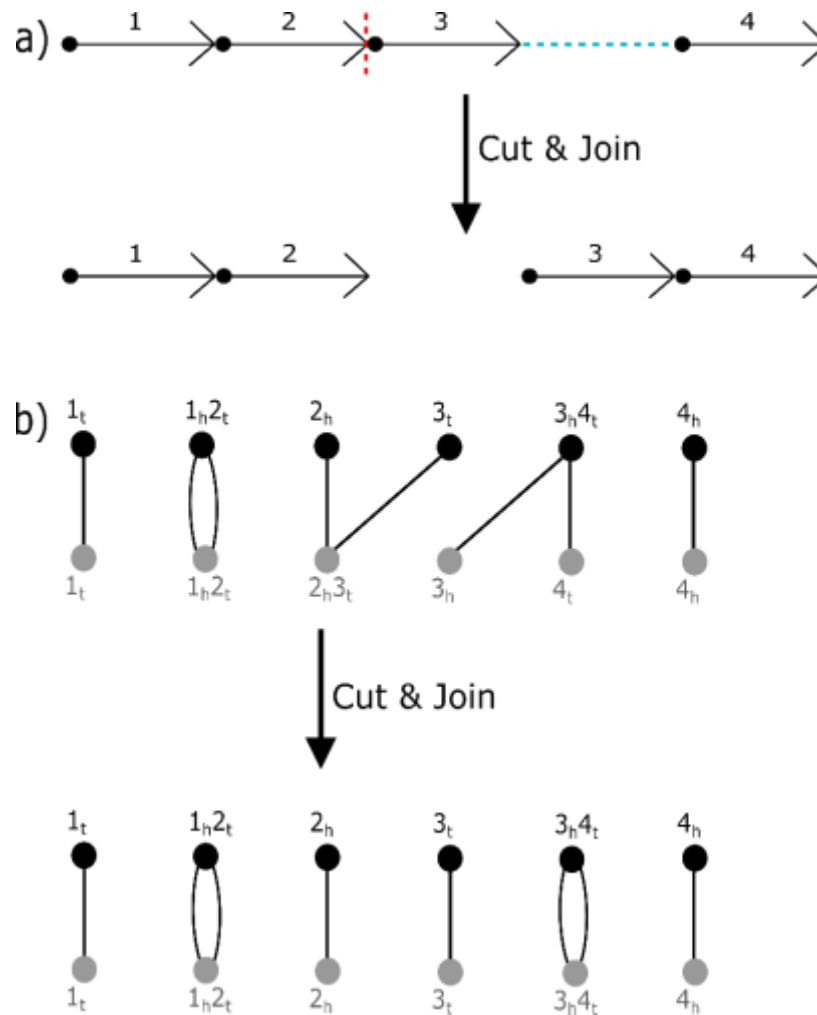


Fig 9. a) Cut-and-join operation. A cut-and-join operation is applied to the genome $G_A = \{(\circ 1, 2, 3 \circ) (\circ 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 4 \circ) (\circ 3, 2 \circ)\}$. The cut component of the operation is represented by the vertical dashed line and the join component by the horizontal dashed line. b) The corresponding adjacency graphs.

This model also makes use of the adjacency graph to address the *genomic sorting problem*. For any pair of genomes, G_A and G_B , the sorting algorithm uses one of the three operations (cut, join or cut-and-join) to decrease the distance between the genomes by one unit for every path and cycle. Given that $G_A \neq G_B$, the adjacency graph of the genomes has *at least* either one path with a length greater than one or a cycle with a length greater than two.

As with the previous model, a linear time algorithm for solving the genomic sorting problem with this model exists [58].

1.4.6.6. Double-cut and join model

Similar to both the SCOJ and SCAJ models, the double-cut-and-join (DCJ) model cuts adjacencies between gene extremities and creates new adjacencies. Unlike the previous two models, this model allows two adjacencies to be removed and two new adjacencies to be formed in a *single operation*. As a result, rearrangement events can be modeled in a fewer number of steps. A simple example of a DCJ operation is given in Fig 10.

The DCJ model was introduced by Yancopoulos et al. [59], and subsequently refined [55]. This model has become the prevalent model for calculating both the genomic distance between genomes and finding an optimal solution to the sorting problem [60].

A DCJ operation involves the breaking of two adjacencies and joining to form two new adjacencies. This model can account directly for inversions, translocations, and chromosome fusions and fissions. Transpositions and block interchanges can be accounted for indirectly, requiring two DCJ operations. These two DCJ operations result in the formation of a circular, intermediary chromosome that is linearized and reinserted in a second DCJ operation.

Fig 11 below illustrates how a DCJ operation may act upon gene adjacencies and telomeric ends. Vertices in the figure represent the gene adjacencies and telomeric ends. If the vertex is external (i.e. a telomeric end) it consists of a single gene extremity. Alternatively, if a vertex is internal, representing a point of adjacency between two genes, it consists of two gene extremities – one from each of the adjacent genes. The gene extremities in the figure are given the labels p , q , r and s and can be representative of either the head or the tail of a gene.

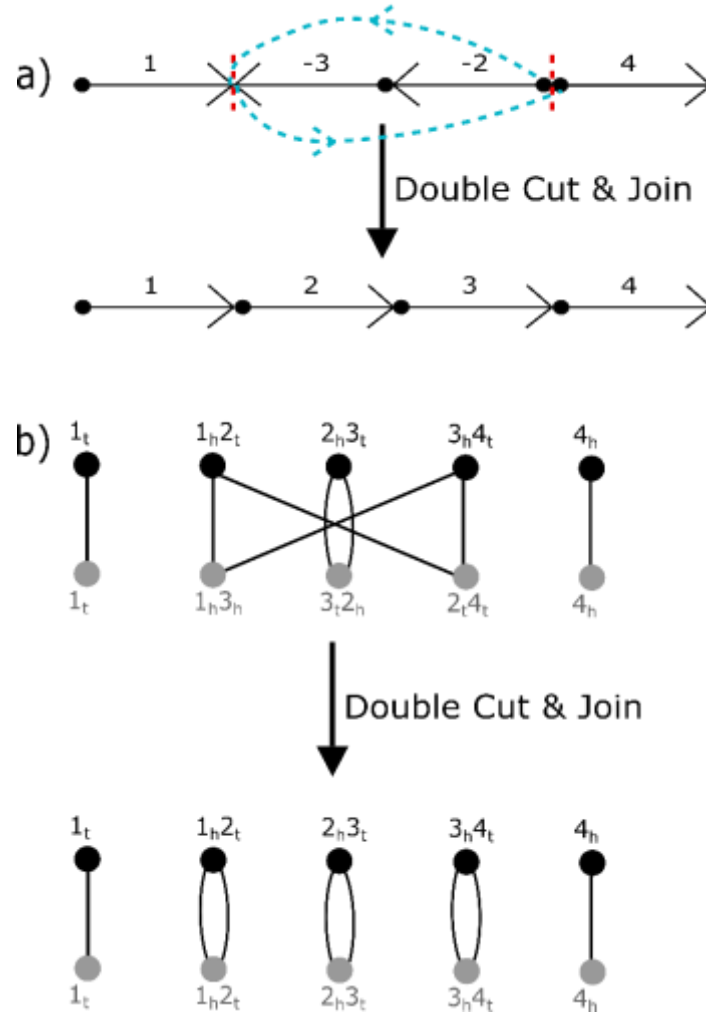


Fig 10. a) Double-cut-and-join operation applied to the genome $G_A = \{(\circ 1, -3, -2, 4 \circ)\}$ to produce the genome $G_B = \{(\circ 1, 2, 3, 4 \circ)\}$. The cut components of the operation are represented by the vertical dashed lines and the join components by the horizontal stippled lines. b) The matching adjacency graph.

The gene adjacencies and/or telomeric ends being acted upon are denoted by the vertices u and v . A DCJ operation can act on these vertices in the following ways:

- If both u and v have are internal vertices, i.e. represent adjacencies, so that $u = \{p, q\}$ and $v = \{r, s\}$, they can be replaced either by $\{p, r\}$ and $\{q, s\}$ or the vertices $\{p, s\}$ and $\{q, r\}$. (Fig 11a).
- If u is internal, $u = \{p, q\}$ and v is external (i.e. represents a telomere), $v = \{r\}$, they can be replaced by $\{p, r\}$ and $\{q\}$ or $\{q, r\}$ and $\{p\}$. (Fig 11b).

- c) If both u and v are external so that $u = \{p\}$ and $v = \{r\}$, they are replaced by $\{p, r\}$. (Fig 11c).
- d) An internal vertex $\{p, r\}$ is replaced by two external vertices $\{q\}$ and $\{r\}$. (Fig 11d).

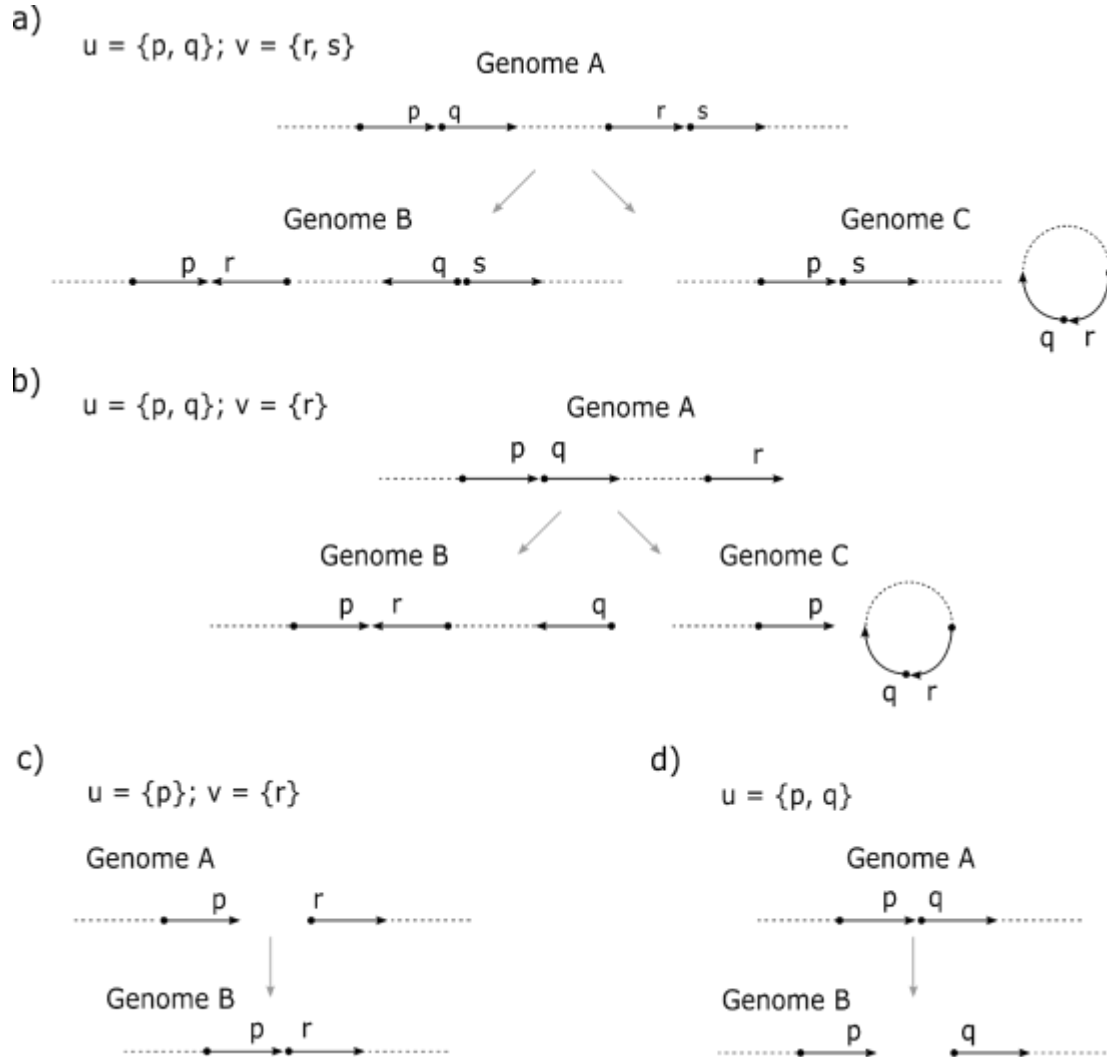


Fig 11. Different ways in which the cut-and-join operation can act on vertices. a) The internal vertices $u = \{p, q\}$ and $v = \{r, s\}$ can be replaced by $\{p, r\}$ and $\{q, s\}$ resulting in an inversion, or by the vertices $\{p, s\}$ and $\{q, r\}$, resulting in an excision and circularization. b) The internal vertex $u = \{p, q\}$ and external vertex $v = \{r\}$ can be replaced by $\{p, r\}$ and $\{q\}$ resulting in an inversion or by the vertices $\{p\}$ and $\{q, r\}$ resulting in breaking off and circularization of an end of a chromosome. c) The external vertices $u = \{p\}$ and $v = \{r\}$ are replaced by the internal vertex $\{p, r\}$, resulting in a chromosome fusion. d) The internal vertex $\{p, r\}$ is replaced by the external vertices $\{p\}$ and $\{r\}$, resulting in a chromosome fission. u and v represent either an adjacency between two genes or a telomeric end. p, q, r and s represent the gene extremities making up the adjacencies and telomeric ends.

Note that the genes in the figure are given an orientation by including arrowheads – this was done merely to improve the clarity of the figure. A DCJ operation can act on any adjacency or telomere irrespective of the types of gene extremities constituting it (i.e. whether the extremities represent the head or tail of genes).

The algorithm presented by [59] made use of the breakpoint graph structure. It had a quadratic runtime and functioned on the constraint that the reincorporation of circular, intermediary chromosomes directly follows its excision, thereby ensuring the correct modeling of transposition and block interchange operations [61].

The refined model by [55] boasted an improved linear runtime compared to the original model, making use of an adjacency graph. The model ignored the original constraint of immediate reincorporation of intermediary circular chromosomes [61]. In permitting the formation of multiple of these intermediary circular chromosomes during any stage of the sorting process, however, the algorithm becomes inapplicable to genomes consisting of only linear chromosomes. From a biological perspective, this presents a problem as higher eukaryotic organisms' genomes typically consist only of linear chromosomes [61].

In 2011, [61] presented an algorithm based on the DCJ model that adheres to the restriction of immediate reincorporation of intermediary circular chromosomes, often referred to as the restricted DCJ model, with a logarithmic time complexity $O(n \log(n))$, where n is the number of shared genes, an improvement to the algorithm introduced in [59]. Their work aims at addressing various genome rearrangement problems applicable to multi-chromosomal linear genomes.

Fig 12 shows an optimal sorting scenario for the genomes below under a) the unrestricted and b) the restricted DCJ model.

In both Fig 12a and Fig 12b genome G_B is transformed into genome G_A , where

$$G_A = \{(\circ 1, 2, 3, 4, 5, 6, 7 \circ)\}, \text{ and}$$

$$G_B = \{(\circ 1, 4, 3, 6, 2, 5, 7 \circ)\}$$

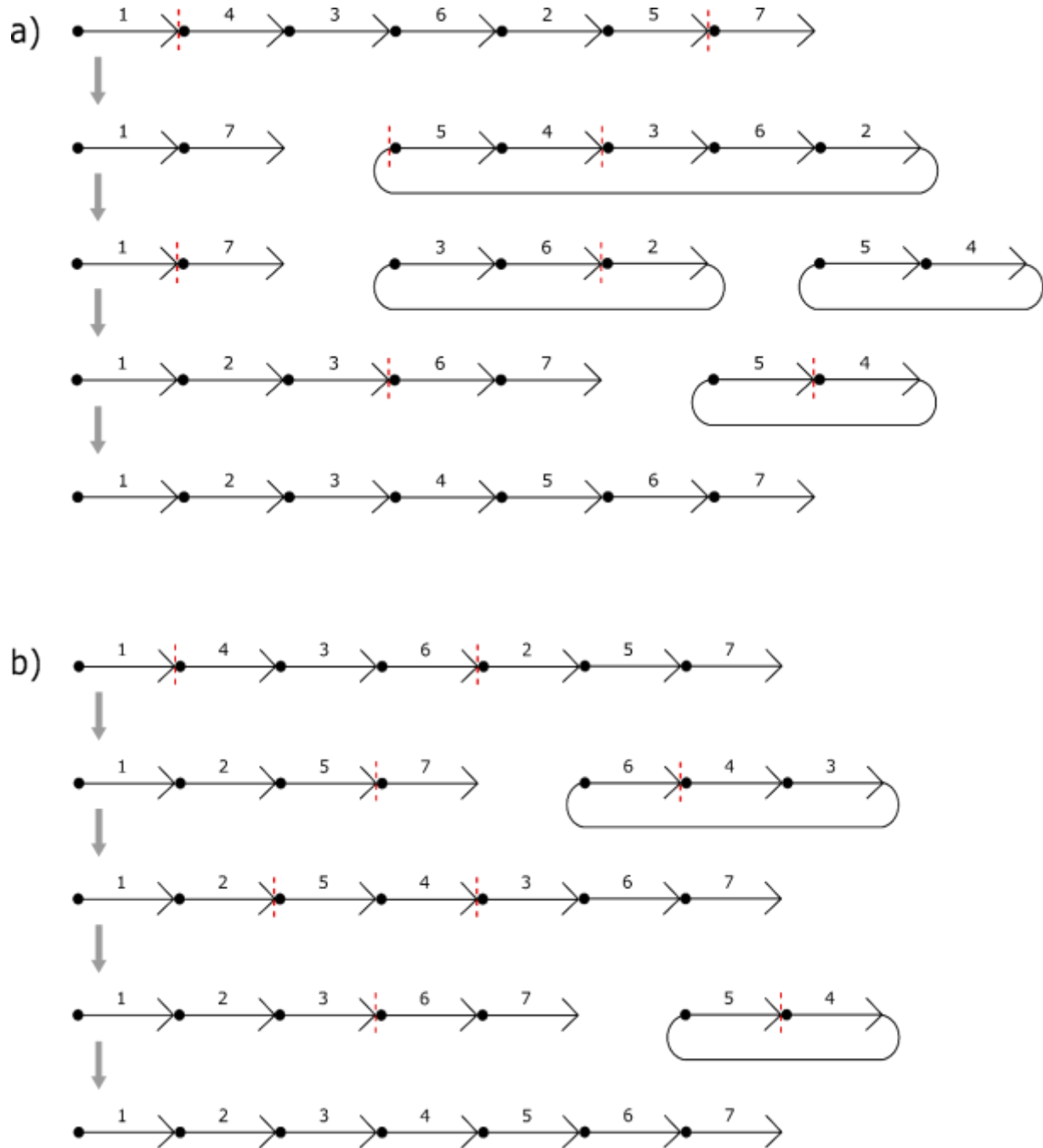


Fig 12. Optimal sorting scenarios for sorting genome $G_B = \{(\circ 1, 4, 3, 6, 2, 5, 7 \circ)\}$ into genome $G_A = \{(\circ 1, 2, 3, 4, 5, 6, 7 \circ)\}$. The genomic distance between the genomes is 4 as both sorting scenarios comprise four steps. The red dashed lines show which adjacencies will be broken in that step. a) an optimal sorting scenario when sorting by general DCJ operations. Note the presence of multiple circular chromosomes during certain stages of the transformation process. b) an optimal sorting scenario when sorting by restricted DCJ operations.

1.5. Bioinformatic tools for the study of large-scale genomic evolution

Since the study of genome rearrangements commenced, a number of genome comparison tools have been developed, enabling the analysis of genomic sequences in the presence of rearrangements. Alignment tools such as Mauve [62] can align two or more genomic sequences in the presence of rearrangements and can be used to identify homologous sequence regions between genomes that differ in location [62]. Genome aligners look at genome rearrangements as isolated events that require identification in order to generate an accurate alignment [62]. Understanding the process by which large-scale genomic evolution occurs requires more than identifying the *location* of rearrangement events.

Other tools for the estimation of evolutionary distance between genome have also been developed. The evolutionary distance is calculable by determining the minimum number of rearrangements required to transform one genomic sequence into another [60,63]. The types of rearrangements used for genome transformation is dependent on the model employed by the tool. The two most well-known open-source bioinformatics tools developed for this purpose is GRIMM [63] and UniMoG [60].

GRIMM utilizes the Hannenhalli and Pevzner (HP) model, which permits inversion, translocations, fusions and fissions [27]. In addition to an estimate of evolutionary distance, GRIMM also generates a single possible sorting scenario which consists of a series of rearrangements able to transform one genome into another (Fig 13).

Though useful for the calculation of the evolutionary distance between genomes, GRIMM's utility as a tool with which to analyse the process of genomic evolution is marred both by (i) its inability to output more than one of an often-vast number of equiprobable sorting scenarios and (ii) by the HP model's inability to account for transposition events.

In contrast to the HP model, the double-cut-and-join (DCJ) model [59] is able to account for all the types of rearrangement events known to occur during the evolutionary process. This model has become the prevalent model [60] and has been implemented in the bioinformatics tool UniMoG to calculate evolutionary distance between two genomes [60]. This tool offers a wide variety of models including the HP, DCJ and rDCJ model.

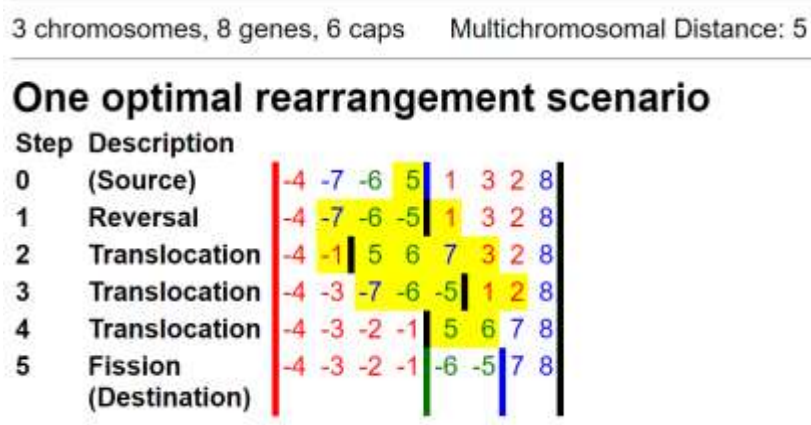


Fig 13. Sorting scenario generated by the GRIMM tool for the transformation of the source genome $[[1, 3, 2, 8], [-5, 6, 7, 4]]$ into the target genome $[[1, 2, 3, 4], [5, 6], [7, 8]]$. The colouring of the integers representing the sequence blocks correspond to the chromosomes on which they are located in the target genome. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.

As in the case of GRIMM, UniMoG outputs a single rearrangement scenario. In contrast to GRIMM, this scenario is not depicted as a series of rearrangement events but a series of DCJ operations (Fig 14). The rearrangement scenario depicted by UniMoG shows a series of steps without identifying the type of rearrangement event represented by each step which makes analysis of the scenario slightly more challenging.

Studying a single or even several of a vast number of equiprobable sorting scenarios is not sufficient to conduct a comprehensive study of the process of genomic evolution. A comprehensive study of the process by which one genome may have evolved into another, requires (i) a model for genome sorting supporting all types of rearrangement events known to occur in nature and (ii) an algorithm that uses this model to identify *all* equiprobable series of rearrangement events that can describe the evolution of one genome into another. Such a tool would facilitate the study of large-scale genomic evolution. The purpose of the MSc project was to develop such a tool.

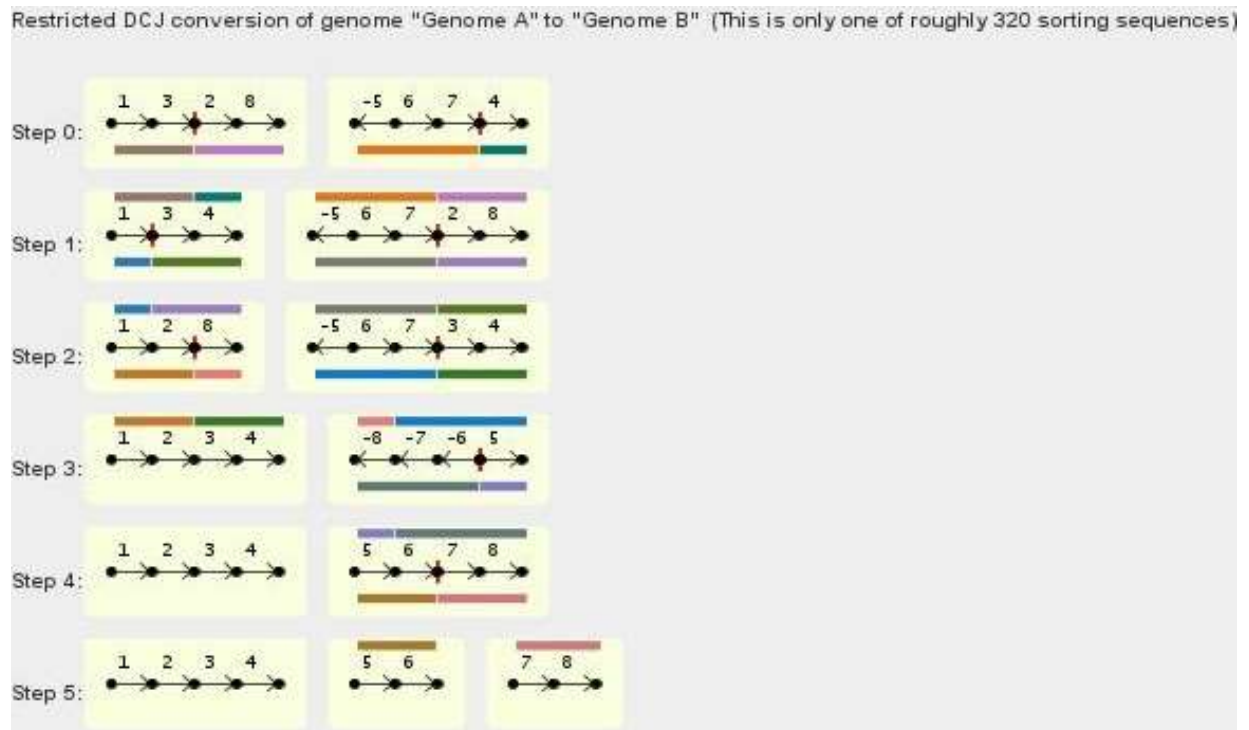


Fig 14. The sorting scenario generated by UniMoG for sorting the same source and target genomes as in Fig 13. The colouring under sequence blocks in one step corresponds to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.

Chapter 2: The development of Genolve

2.1. Brief description of the tool

For this MSc project, the tool ‘Genolve’ was developed to facilitate the study of large-scale genomic evolution. The tool takes as input two sets of labeled homologous genes representing two evolutionary related genomes. Using the rDCJ model, the one set of genes is transformed into the other by applying successive DCJ operations. The rDCJ model is utilized by a recursive algorithm to ensure the identification of *all* series of rearrangement events able to achieve this transformation.

2.2. Introduction: The rDCJ model and its limitations

Genome rearrangements enable us to study the evolutionary process on a genome-wide, structural level. Solving the genomic sorting problem takes us one step closer to understanding which set of rearrangement events resulted in the evolution of one genome into another. Of the models and algorithms addressing this problem, the best suited to account for *all* of the types of rearrangement events that may occur during the evolutionary process, is the restricted double-cut-and-join (DCJ) model [55,59]

2.2.1. The DCJ operation

As was stated in the introductory section of this thesis, A DCJ operation involves the breaking of two adjacencies to form two different adjacencies (see Fig 11 for the different ways in which a DCJ operation can act upon gene adjacencies and telomeric ends).

A single DCJ operation is necessary to model inversions, translocations (balanced and unbalanced) and chromosome fusions and fissions (Fig 1a-e), allowing these types of biological rearrangement events to be accounted for directly under the DCJ model and its derivatives. In addition to these rearrangements, the model can also account for transpositions, inverted transpositions and block interchanges, albeit indirectly. These types of events require two distinct DCJ operations, the first of which results in the formation of a circular chromosomal intermediate. A second DCJ operation, entailing the linearization and reincorporation of this intermediate elsewhere in the genome, is required to complete these types of rearrangement events (Fig 1f-h).

2.2.2. The restricted DCJ model: an improvement on the DCJ model

The formation of circular chromosomes during the evolution of linear genomes is unlikely. As has been pointed out, the circular chromosomal intermediates formed under the DCJ model are nonetheless necessary structures to allow the model to perform transpositions and block-interchanges. The DCJ model does not, however, place any restriction on *when* circular intermediates need to be reincorporated, nor on *how many* are permitted to co-exist. A derivative of this model, the restricted DCJ model does. Under this model, linearization and reinsertion of a circular intermediate proceed immediately after its formation. This restriction ensures that a transposition or block-interchange event be completed prior to the execution of another rearrangement. Consequently, from a biological perspective, the rDCJ model is considerably more accurate in modeling linear genomic evolution than its unrestricted counterpart. (See Fig 12 for a sorting scenario for the same genomes under the DCJ and rDCJ models, respectively.)

2.2.3. Problems with the rDCJ model

Despite the improved performance of the rDCJ model relative to the DCJ model, there are still several limiting factors that need to be considered when evaluating the biological accuracy and applicability of the model with reference to the identification of possible evolutionary events that would have resulted in the evolution of one genome into another.

2.2.3.1. A single solution

The genomic sorting problem, and those models addressing it, including the DCJ and rDCJ models, concerns the identification of a *single* optimal sorting scenario. Yet, there are often numerous different series of rearrangement events that could describe the transformation of one genome into another. Analysis of only one of multiple equiprobable scenarios would not be sufficient for drawing biological conclusions regarding the evolutionary relationship between two genomes [64,65]. In order to gain a better understanding of underlying patterns and mechanisms of genome evolution, the entire set of optimal solutions must be considered. An algorithm capable of identifying and reporting all optimal (most parsimonious) solutions is thus required.

2.2.3.2. Operation cost

Despite the improvement of the rDCJ model in terms of the restriction it places on the reincorporation of circular chromosomal intermediates, the use of these structures still presents a problem.

If we assume each *DCJ operation* has an operational cost of one, then those *rearrangement events* that can be modeled in a single DCJ operation (*i.e.*, inversions, translocations and chromosome fusions and fissions) will each have a corresponding operational cost of one. In contrast, transpositions and block-interchanges, both of which require two DCJ operations for their execution, will have a total operational cost of two.

An algorithm capable of identifying all most parsimonious solutions will output those paths with the lowest total operational cost. If the cost of all the rearrangements is not equivalent, these results will be inaccurate.

The figure below shows two sorting scenarios for transforming the genome $[[1, 2, 3], [6, 7, 8, 4, 5, 9, 10]]$ into the genome $[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]$. In Fig 15a (*operation cost fig*) two unbalanced translocations are used for the transformation, each of which requires a single DCJ operation and thus the total cost of the transformation is two. Fig 15b (*operation cost fig*) shows how the transformation can occur using a single intrachromosomal transposition operation. Given that this type of rearrangement event requires two DCJ operations, the total cost of this sorting scenario will also be two. The algorithm tasked with finding all most parsimonious solutions will thus give both of these sorting scenarios as output despite the fact that Fig 15a requires two rearrangement events and Fig 15b only one.

Addressing this problem will result in a more accurate output from a biological perspective and, in numerous cases, decrease the solution space, *i.e.*, the number of sorting scenarios reported.

2.2.3.3. Block-interchanges

Block-interchanges are permitted by the DCJ model and its derivatives. Though useful from a computational viewpoint, this type of event has not been identified as a common type of rearrangement event during genomic evolution [66]. Allowing block-interchanges will thus have a negative impact on the biological accuracy of a model.

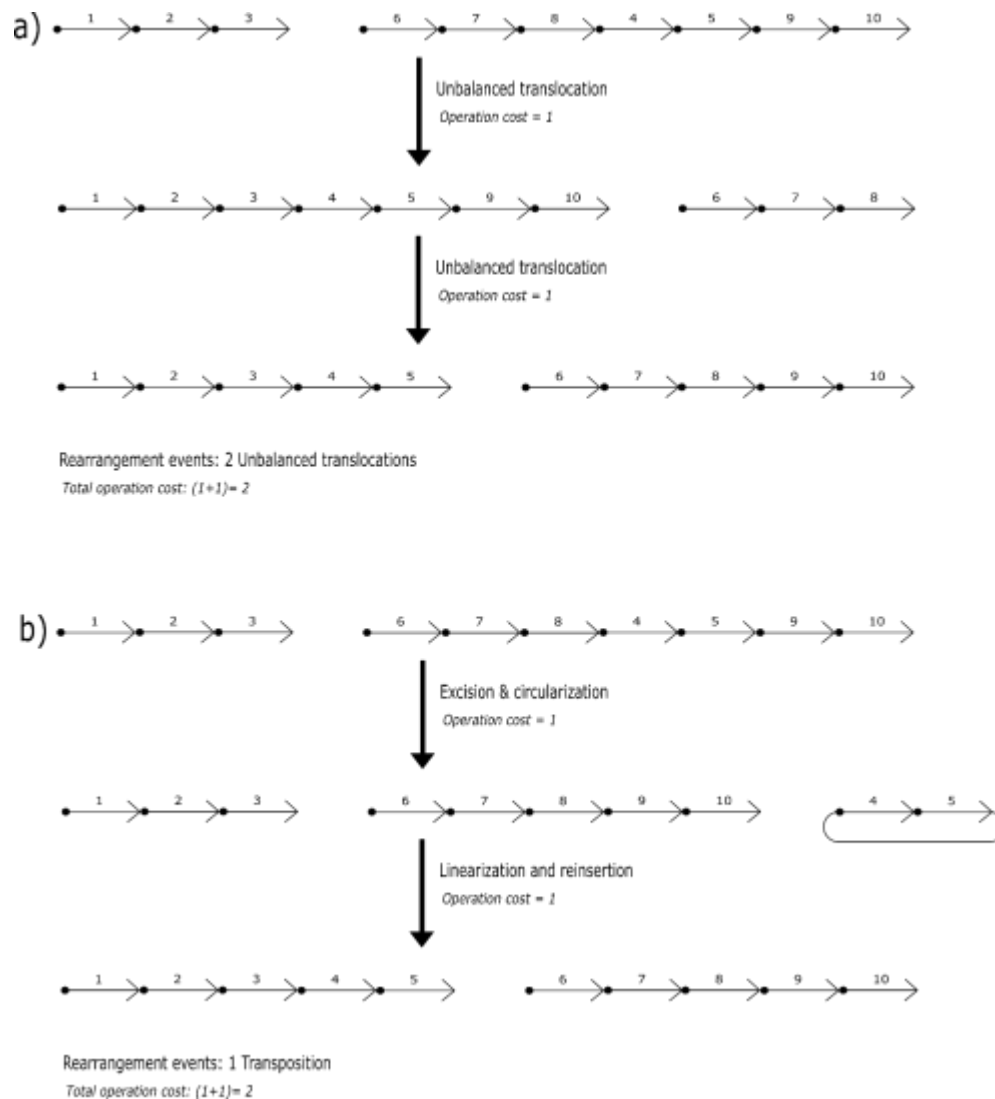


Fig 15. Depiction of the sorting scenario for transforming the genome $[[1, 2, 3], [6, 7, 8, 4, 5, 9, 10]]$ into the genome $[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]$ using two different series of rearrangements a) Two successive unbalanced translocations are used for the transformation b) A single transposition, consisting of an excision and circularization step followed by a linearization and reinsertion step, is used for the transformation.

2.2.3.4. Identification of rearrangement events from DCJ operations

The DCJ model and its derivatives can account for all the types of rearrangement events. These events are, however, represented as one or more DCJ operations. The output of tools that offer this model generally reports sorting scenarios as a series of DCJ operations, not as a series of *rearrangement events*. In developing a tool specifically to advance the study of genomic evolution

and rearrangement events, tailoring the output to report actual rearrangement events and not only DCJ operations used to execute them would be more useful.

2.2.3.5. Incorporation of a weighting system

There is evidence to suggest that some rearrangements occur in higher frequencies than others in certain species. Thus, it would be appropriate to allow for a user-adjustable set of relative frequencies in which the various types of rearrangements are expected to occur for the set of input genomes. Incorporating a weighing system that uses these frequencies to make low-frequency rearrangements more costly than high-frequency rearrangements may increase the likelihood that the ‘true’ solution or series of rearrangements events describing the evolution of one genome into another is present in the output of the tool.

2.3. Development of the tool

The development of the ‘Genolve’ software tool occurred in distinct phases. The following section will thus be organized as follows: First, a brief introduction to the notation used for the remainder of this thesis will be given. I will then report the methodology for each respective phase and report and discuss results where appropriate.

The first phase of the project was simply generating the Python code to implement the rDCJ model. Each of the problems relating to the rDCJ model mentioned in the previous section was addressed in Phase II – VI respectively. The algorithmic efficiency and runtime were investigated and consequently improved in Phase VII.

2.3.1. Notation and terminology

Prior to continuing, it is necessary to revisit some of the notation presented in the first section of this thesis, new notation and terminology will also be introduced. Note that some of the notation will differ slightly from that used up to this point. The notation used in earlier parts of this thesis (specifically in the review of the literature) was selected to enhance the understandability of the concepts that were being explained. From this point on, notation is selected to correspond with

that which has been used in the code files of the Genolve tool in the hope that it makes the content of those files easier to follow.

A *sequence block* is a segment of DNA with a tail (x), followed by a head (x.5), called *extremities* (for inverted sequence blocks x.5 precedes x), where x is an element of the set of Natural numbers. A *genome* for a set of sequence blocks is a set of adjacent extremities where each extremity is contained in a maximum of one adjacency, i.e. the end of one sequence block can be adjacent to at most the end of any one other sequence block. If an extremity is not contained within an adjacency, it is located on the end of a linear chromosome and represents a telomeric end.

Thus, a *genome* consists of one or more numerically labeled *sequence blocks* contained within one or more linear or circular chromosomes. If a sequence block is in the inverted orientation, i.e. its head precedes its tail, the numerical label of the block is a negative integer, else it is a positive integer.

Sequence blocks that lie on the same chromosome are contained within a set of square brackets ('[]'). If the chromosome is circular, the first element in the brackets is 'o'. The set of chromosomes that form part of a genome are themselves contained within a set of square brackets representing the genome.

The notation for the genome graph in Fig 5 will thus be:

$$G = [[3,-2,-4,-6,5,1],[\text{'o'},10,8,-7,-9]]$$

A *list of adjacencies* or an *adjacency list* of a genome is a list containing all *extremities* representing telomeric ends (arranged from small to larger) followed by all the adjacencies. Each pair of extremities per adjacency are arranged in increasing order, and the set of all adjacencies are arranged in a similar fashion – based on the value of the first extremity in each adjacency.

Below is the adjacency list for the genome in Fig 5.

$$AL_G = [1.5, 3, (1, 5.5), (2, 4.5), (2.5, 3.5), (4, 6.5), (5, 6), (7, 9.5), (7.5, 8.5), (8, 10.5), (9, 10)]$$

2.3.1.1. *Source, target and intermediary genomes*

A *source* and *target genome* are a pair of evolutionary related genomes consisting of DNA segments that share homology between the genomes. The *source genome* can be transformed into the *target genome*, by application of a series of DCJ operations.

The DNA segments (sequence blocks) making up the respective genomes are given numerical labels so that the label of a block in the *source genome* corresponds to its homologous counterpart in the *target genome*.

The numerical labels of the sequence blocks in the *target genome* occur in consecutive order. A *source genome* must have at least one sequence block present at a different position relative to its homologous counterpart in the *target genome*.

Once a series of one or more DCJ operations have been applied to the *source genome* so that positions of all sequence blocks correspond to their homologous counterparts in the *target genome*, the genome transformation process is complete.

If more than one DCJ operation is required to transform the *source genome* into the *target genome*, they will be separated by a series of *intermediary genomes*. After applying of each DCJ operation a new *intermediary genome*, will be generated, with its own, unique set of chromosomes and list of adjacencies.

2.3.1.2. *Final- and non-final- state adjacencies and telomeres*

All the adjacencies between the sequence block extremities in a target genome are termed *final state adjacencies* and all telomeric ends in a target genome, *final state telomeres*. Any adjacency or telomeric end not present in the target genome are *non-final state adjacencies* or *non-final state telomeres*, respectively.

While the transformation of a source genome is not yet complete, it will consist of a number of *non-final state adjacencies* and/or *telomeres* and zero or more *final state adjacencies* and/or *telomeres*. Once the number of *non-final state adjacencies* and *telomeres* in the source genome equates to zero, transformation into the target genome is complete.

2.3.2. I: Initial coding of the rDCJ model

Phase I of the project entailed (i) encoding the rDCJ model described in [67] and (ii) ensuring it functioned correctly. The model was coded in Python3 and tested using simulated data of which the solutions were known. The source code of rDCJ implementation is available from <https://github.com/HFouch/GenolveFinal>.

Briefly, the model works as follows:

Input: a source genome and a target genome

For each input genome:

 Generate a *list of adjacencies*

While the source/intermediary genome is not equal to the target genome:

1. Iterate through the list of final state adjacencies (i.e. the target genome adjacency list) until an extremity or adjacency not present in the source/intermediary genome is found
2. Execute the DCJ operation required to create the extremity or adjacency in the source genome
3. Record the DCJ operation and resulting intermediary genome
4. If the resulting intermediary genome contains a circular chromosome:
 - 4.1. Find a non-final state adjacency present in the circular chromosome of which a corresponding partner exists within a linear chromosome
 - 4.2. Execute a DCJ operation involving these two adjacencies
 - 4.3. Record the DCJ operation and resulting intermediary genome

Return the series of DCJ operations and intermediary genomes

Below is an example of how this model will execute for the following pair of input genomes: target genome = [[1, 2, 3, 4, 5, 6]] and source genome = [[1, -2, 3, 5, 4, 6]].

INPUT

target_genome = [[1, 2, 3, 4, 5, 6]]

source_genome = [[1, -2, 3, 5, 4, 6]]

#Generate adjacency lists

```
target_adjacencies = [1, 6.5, (1.5, 2), (2.5, 3), (3.5, 4), (4.5, 5), (5.5, 6)]
```

```
source_adjacencies = [1, 6.5, (1.5, 2.5), (2, 3.5), (3.5, 5), (4, 5.5), (4.5, 6)]
```

#1. Iterate through the target_adjacency list until an element not present in source_adjacencies is found

```
#The first element in target_adjacencies but not in source_adjacencies is (1.5, 2)
```

```
element = (1.5, 2)
```

#2. Execute the required DCJ operation to create the element

```
# The DCJ operation to create (1.5, 2) involves the following two adjacencies in the source_genome: (1.5, 2.5) and (2, 3).
```

```
adjacency_1 = (1.5, 2.5)
```

```
adjacency_2 = (2, 3)
```

```
new_adjacency_1 = (1.5, 2)
```

```
new_adjacency_2 = (2.5, 3)
```

```
source_adjacencies.remove(adjacency_1)
```

```
source_adjacencies.remove(adjacency_2)
```

```
source_adjacencies.append(new_adjacency_1)
```

```
source_adjacencies.append(new_adjacency_2)
```

```
adjacencies_destroyed = (adjacency_1, adjacency_2)
```

```
adjacencies_formed = (new_adjacency_1, new_adjacency_2)
```

```
#use adjacency list to calculate new genome
```

```
intermediary_genome = [[1, 2, 3, 5, 4, 6]]
```

#3. Record DCJ operation and intermediary genome

```
list_of_DCJ_operations.append((adjacencies_destroyed, adjacencies_formed))
```

```
list_of_intermediary_genomes.append(intermediary_genome)
```

#4. Does the intermediary genome contain circular chromosomes?

```
No
```

```
intermediary_genome does not equal target_genome
```

Generate new adjacency list for intermediary genome


```
Intermediate_adjacencies = [1, 6.5, (1.5, 2), (2.5, 3), (3.5, 5), (4, 5.5), (4.5, 6)]
```

#2. Iterate through the target_adjacency list until an element not present in source_adjacencies is found

#The first element in target_adjacencies but not in intermediate_adjacencies is (3.5, 4)

```
element = (3.5, 4)
```

#3. Execute the required DCJ operation to create the element

The DCJ operation to create (3.5, 4) involves the following two adjacencies in the intermediary_genome:(3.5, 5) and (4, 5.5).

```
adjacency_1 = (3.5, 5)
```

```
adjacency_2 = (4, 5.5)
```

```
new_adjacency_1 = (3.5, 4)
```

```
new_adjacency_2 = (5, 5.5)
```

```
intermediate_adjacencies.remove(adjacency_1)
```

```
intermediate_adjacencies.remove(adjacency_2)
```

```
intermediate_adjacencies.append(new_adjacency_1)
```

```
intermediate_adjacencies.append(new_adjacency_2)
```

```
adjacencies_destroyed = (adjacency_1, adjacency_2)
```

```
adjacencies_formed = (new_adjacency_1, new_adjacency_2)
```

#use adjacency list to calculate new genome

```
intermediary_genome = [[1, 2, 3, 4, 6], ['o', 5]]
```

#3. Record DCJ operation and intermediary genome

```
list_of_DCJ_operations.append((adjacencies_destroyed, adjacencies_formed))
```

```
list_of_intermediary_genomes.append(intermediary_genome)
```

#4. Does the intermediary genome contain circular chromosomes?

Yes

#4.1 Find a non-final state adjacency present in the circular chromosome for which a corresponding partner exists within a linear chromosome

```
Intermediate_adjacencies = [1, 6.5, (1.5, 2), (2.5, 3), (3.5, 4), (4.5, 6), (5, 5.5)]
```

circular_adjacency = (5, 5.5)

The final state adjacency in which 5 occurs is (4.5, 5) thus a potential partner adjacency for (5, 5.5) would be one that contains the extremity 4.5.

final_state_adjacency = (4.5, 5)

partner_adjacency = (4.5, 6) *#(4.5, 6) does occur within a linear chromosome making it a suitable partner*

#4.2 Execute the required DCJ operation to create the element

The DCJ operation to destroy (5, 5.5) involves the following two adjacencies: (5, 5.5) and (4.5, 6) in the intermediary genome

adjacency_1 = circular_adjacency

adjacency_2 = partner_adjacency

new_adjacency_1 = final_state_adjacency

new_adjacency_2 = (5.5, 6)

intermediate_adjacencies.remove(adjacency_1)

intermediate_adjacencies.remove(adjacency_2)

intermediate_adjacencies.append(new_adjacency_1)

intermediate_adjacencies.append(new_adjacency_2)

adjacencies_destroyed = (adjacency_1, adjacency_2)

adjacencies_formed = (new_adjacency_1, new_adjacency_2)

intermediary_genome = [[1, 2, 3, 4, 5, 6]]

#4.3 Record DCJ operation and intermediary genome

list_of_DCJ_operations.append((adjacencies_destroyed, adjacencies_formed))

list_of_intermediary_genomes.append(intermediary_genome)

intermediary_genome is equal to the target_genome

Return the DCJ operations and intermediary genomes

list_of_DCJ_operations = [[[1.5, 2.5), (2, 3), (1.5, 2), (2.5, 3)], [(3.5, 5), (4, 5.5), (3.5, 4), (5, 5.5)], [(5, 5.5), (4.5, 6), (4.5, 5), (5.5, 6)]]

list_of_intermediary_genomes = [[[1, 2, 3, 5, 4, 6]], [[1, 2, 3, 4, 6], ['o', 5]], [[1, 2, 3, 4, 5, 6]]]

2.3.3. II: *The single solution problem*

The simple application of the DCJ model generates a single series of DCJ operations which, when applied to the source genome would allow for its transformation into the target genome. To gain a deeper understanding of the process of genomic evolution, it is necessary to study and analyze *all* most parsimonious series of DCJ operations that result in complete genome transformation. Solving the single solution problem thus required developing an algorithm that uses the DCJ model to find all possible series of operations capable of transforming the source genome into the target genome, and outputting the most parsimonious of these operation series.

To accomplish this, the entire set of possible DCJ operations for the source genome and all intermediary genomes that occur between the source and target genome needs to be calculated and executed. It is also necessary to keep track of how the source genome and various intermediary genomes are connected so that paths starting from the source genome, visiting one or more intermediary genomes and terminating at the target genome can be extracted. For each path from source to target, a list of the DCJ operations taken and each of the intermediary genomes their execution resulted in would be required.

This problem is not a simple one. Just as for the source genome, each intermediary genome has its own set of DCJ operations that need to be executed which will, in turn, generate another set of intermediates. This continues until the final ‘lowest level’ intermediates (furthest away from the source genome) are transformed (by applying a single DCJ operation) into the target genome.

Calculating all the possible paths from the source genome to the target genome (whilst storing all DCJ operations and intermediary genomes in memory) can thus start to take up a fair amount of storage space. However, it is possible that two or more different series of rearrangements result in the same intermediary genome somewhere *en-route* to the target genome. From that shared intermediary genome onwards, the DCJ operations executed and intermediary genomes visited in order to reach the target genome would be identical.

Therefore, it would be inefficient to calculate the path from the shared intermediate to the target genome numerous times. Instead, the algorithm should recognize that it has encountered and ‘solved’ that particular intermediate before and know that all paths encountering it should follow the previously calculated path from that intermediate to the target genome.

In summary, the algorithm that was to be developed to solve the single solution problem thus needed to be able to:

- (i) Calculate and execute DCJ operations for the source and intermediary genomes efficiently.
- (ii) Store the intermediates and DCJ operations in a memory-efficient manner.
- (iii) Calculate the path from an intermediate to the target genome only once, by recognizing if a specific intermediate has been encountered before.
- (iv) Store the DCJ operations and intermediary genomes in such a way that the different paths leading from the source genome to the target genome can be easily extracted.

2.3.3.1. The algorithm

The algorithm that was developed can broadly be divided into three parts:

- I. Generation of a dictionary containing the source genome, target genome and all intermediary genomes.
- II. Utilization of the dictionary to build a directed acyclic network of all paths from the source genome to the target genome.
- III. Identification of the shortest (most parsimonious) paths through the network (from source to target).

I. Dictionary Construction

Creating a dictionary containing the source, target and all intermediary genomes requires that, for each genome which is not the target genome, all possible DCJ operations be identified and executed.

A list of possible DCJ operations for a given genome is generated by determining which final state adjacencies and telomeres are not present in the genome and identifying the DCJ operations which

would result in their formation. The successive execution of each DCJ operation in the list of all possible operations yields a set of intermediary genomes.

The process of finding and executing all possible DCJ operations are repeated for each of the intermediary genomes, only terminating when no more non-final state adjacencies or telomeres exist, i.e. once transformation into the target genome is complete.

This repetition of the same computational tasks, terminating once some requirement is met, led me to consider a recursive approach for constructing the dictionary.

Recursions are often used to solve problems that can be broken up into smaller versions of itself. In this case, the problem is transforming the source genome into the target genome by calculating and following all the various paths via which this may occur. In essence, however, the ‘small’ problem is transforming the source genome into the intermediary genome that directly proceeds it, *i.e.* applying a single DCJ operation to the source genome to generate an intermediary genome that is now slightly closer or more similar to the target genome. The same small problem applies to each of the intermediates. Thus the ‘small’ problem that needs to be solved multiple times is:

For some genome:

Determine whether it is the target genome

If it is not:

1. Find all possible DCJ operations.
2. Execute all possible DCJ operations (to generate the list of intermediates proceeding the genome in question).

For each genome (source or intermediary) it is also necessary to be able to recall what the possible DCJ operations were and which intermediates resulted from their execution.

In order to achieve this, each genome has associated with it a *list of child operations* consisting of the possible DCJ operations and a *list of children* made up of the intermediary genomes that resulted from the execution of these DCJ operations.

As per the specification, the algorithm for dictionary construction also needs to recognize whether a particular intermediary genome has been encountered and avoid recalculating its intermediates. To achieve this, each genome had a unique hash key associated with it. This serves as the dictionary key pointing to a genome and its associated list of child operations and children. The hash key of a genome is dependent on its adjacency list. If two genomes have equivalent adjacency lists (*i.e.*, the genomes are identical), their hash keys will also be identical

Whenever a new intermediary genome x is generated from a genome y by executing a DCJ operation, its hash key is calculated and used to determine whether the genome x is already present in the dictionary:-

The pseudo-code for the recursive function used to construct the dictionary is given below

1	Define the dictionary (create an empty dictionary)
2	Add the source genome and the target genome to the dictionary
3	Make the source genome the <i>current genome</i>
4	Call the recursive function
5	
6	<i>RecursiveFuntion(current genome):</i>
7	For the current genome:
8	Find and create a list of all possible DCJ operations
9	For each DCJ operation:
10	Execute the DCJ operation
11	Add the DCJ operation to the current genome's list of child operations
12	Add the resulting intermediary genome to the current genome's list of children
13	Check if the resulting intermediary genome is already in the dictionary
14	If it is in the dictionary:
15	Move to and execute the next DCJ operation (jump to line 10)
16	Else:
17	Add it to the dictionary
18	Make it the current genome (jump to line 7)

Fig 16 shows how the algorithm will, in a recursive fashion, identify each of the intermediates, enabling it to build the dictionary. The algorithm will start with source genome being set to the *current genome*. After calculating the list of possible DCJ operations ($[a, b, c]$), it will execute the first operation, in this case, a , and add it to the *list of child operations* associated with the source genome.

The resulting genome is Intermediate 1, which is added to the source genome's list of children. Intermediate 1 is not in the dictionary, so it is added to the dictionary and the *current genome* is set to Intermediate 1, after which the recursive function is called once more. Operations d and e are possible from Intermediate 1. The algorithm executes d to produce Intermediate 2 (adding d to Intermediate 1's *list of child operations* and Intermediate 2 to its *list of children*). Intermediate 2 is not yet present in the dictionary and is therefore added to the dictionary and set as the current genome before the recursive function is called. Operation f is the only possible operation for Intermediate 2, and this generates Intermediate 3 (not present in the dictionary). After executing operation g , the target genome is generated from Intermediate 3. The target genome is already present in the dictionary, so the algorithm moves to execute the next operation in Intermediate 3's list of DCJ operations. g was, however, the last (and only) operation possible from Intermediate 3 so the algorithm moves another level up to execute the next operation for Intermediate 2. Again, the final operation in the list, f , has already been executed, so the algorithm moves up another level to Intermediate 1. The next operation possible from Intermediate 1 is e , which is executed.

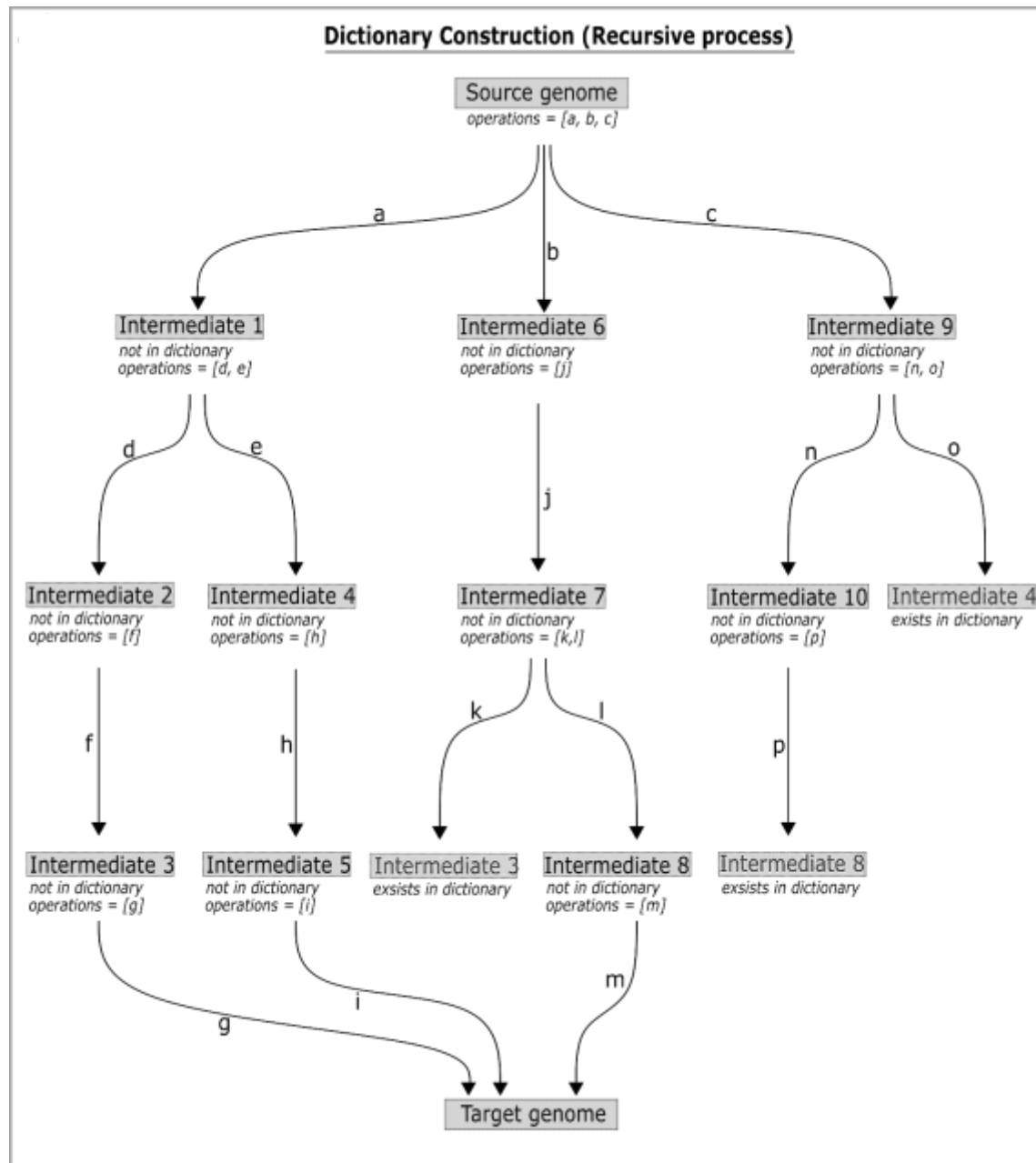


Fig 16. Illustration of how the Genolve algorithm recursively generates a dictionary containing the source, target and all intermediary genomes. In the figure, the process occurs from left to right. With the left-most path being completed before proceeding to the next left-most path. Genomes already present in the dictionary are not added a second time; therefore, paths terminate at the second occurrence of Intermediates 3, 8 and 4, respectively.

This generates Intermediate 4, which gets added to the dictionary. Operation *h* is executed to generate Intermediate 5, followed by operation *i* to generate the target genome. On encountering the target genome (already present in the dictionary) the algorithm will once again continue

moving up one level at a time until it finds a list of DCJ operations where the final operation has not yet been executed. In this case, it will move all the way up to the source genome to execute operation b to generate Intermediate 6. Intermediate 7 is generated from Intermediate 6 by executing j .

There are two DCJ operations possible for Intermediate 7, k and l . The execution of k results in the generation of Intermediate 3. This intermediate is already present in the dictionary. Thus, the algorithm adds it to Intermediate 7's *list of children* and moves to execute the next DCJ operation for Intermediate 7, in this case, l . This process continues until the final DCJ operation in all of the intermediate's and the source genome's list of operations has been executed.

Note that as with Intermediate 3, both Intermediate 8 generated from Intermediate 10 by executing p , and Intermediate 4 generated from Intermediate 9 via the execution of o , is already present in the dictionary and thus the algorithm does not recalculate the DCJ operations and intermediates necessary to reach the target genome from these intermediates.

The recursive 'exploration' of the different paths terminating in the target genome thus occurs from left to right, the left most path being completed before exploring the next path.

Constructing a dictionary containing all the intermediates makes it easy to assess whether the intermediate has been encountered (and 'solved') before, preventing unnecessarily calculating the path from it to the target genome numerous times.

Having a *list of child operations* and a *list of children* associated with the source genome and each of the intermediary genome makes it possible to determine the connections between a genome and each of its children. This is necessary to construct a network of all the paths that lead from the source genome to the target genome.

II. Building the Network of Solutions

The network of solutions or paths from the source genome to the target genome was constructed using the Networkx package in Python. The network is a directed acyclic graph that starts at the source genome and terminates at the target genome. Each of the genomes in the dictionary is added to the graph as a node/vertex. The *list of children* associated with each genome is then used to

generate edges pointing from a node representing a specific genome to each of the nodes representing its children. The *list of child operations* is used to identify which DCJ operations are represented by the various edges. Fig 17 shows the network of solutions for the same pair of source and target genomes used to illustrate the recursive process for dictionary construction in Fig 16.

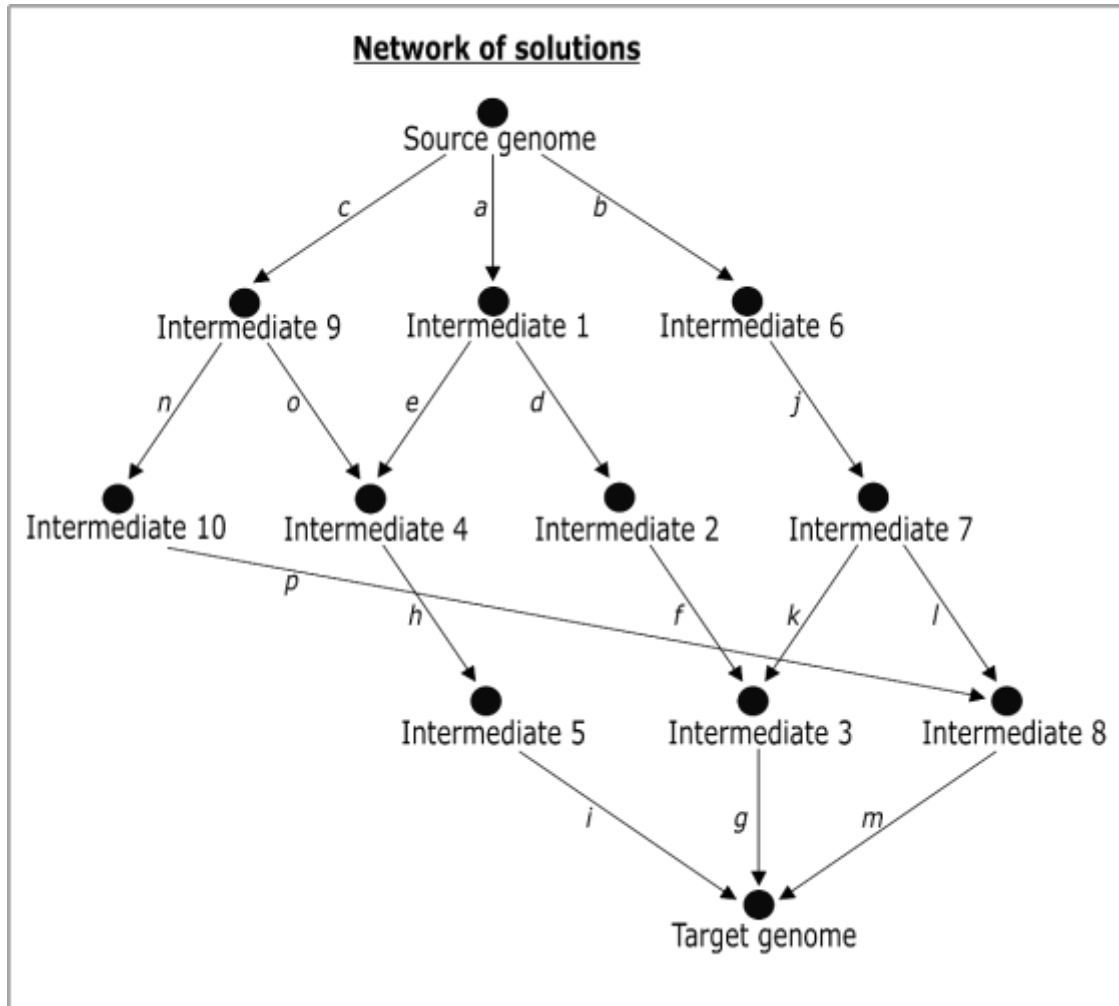


Fig 17. The network of solutions constructed from the genomes present in the dictionary created in Fig 16. The network shows different paths that can be taken to get from the source genome to the target genome, i.e., to achieve the successful transformation of the source into the target genome.

Note that in the network of solutions, both Intermediates 9 and 1 point to Intermediate 4, from which the path to the target genome is the same. The same is true for Intermediate 10 and 7, which both point to Intermediate 8, and Intermediates 2 and 7, which both point to Intermediate 3.

III. Shortest Path Identification

The shortest path problem is by no means novel and has been extensively studied in the field of graph theory. Put simply, it is the problem of finding a path between two vertices in a connected graph such that the cost or weights of the sum of the edges constituting the path is minimized.

There are numerous variations of the problem pertaining to directed and undirected graphs, weighted, unweighted and negatively weighted graphs and single source or all-pairs shortest paths.

Our particular network or graph is a non-negatively weighted directed acyclic graph for which we are interested in finding single-source shortest paths (we are only interested in paths starting from the source genome and terminating at the target genome). The best-suited algorithm to achieve this is Dijkstra's algorithm. This algorithm, available in the Networkx python library, was thus implemented to find all shortest paths through the network.

2.3.4. III: The operation cost problem

The higher operational cost of transpositions and block-interchanges relative to other rearrangements relates to the requirement of two DCJ operations for their execution.

Each edge in the network of solutions represents a single DCJ operation and, by extension, either one inversion, translocation, fusion or fission event or half of a transposition or block-interchange event. In the absence of edge weights, all edges are weighted equally, and a shortest path through the network constitutes one which traverses the minimum number of edges. When edges carry different weights, however, a shortest path through the network becomes one that minimizes the combined weights/cost of the edges it traverses.

The operation cost problem would thus be solvable by altering the weights of certain edges.

The solution that was implemented to solve the operation cost problem was as follows:

Those edges representing transpositions and block-interchanges are assigned a cost of 0.5, whilst the cost of all other edges is set to one. This ensures that the total cost of each type of rearrangement event is equivalent.

In order to accomplish this, an additional list of ‘child weights’ to be associated with each node (in addition to the lists of *children* and *child operations*) was incorporated so that the algorithm now functions as follows:

After the execution of a DCJ operation, the set of chromosomes of the resulting intermediary is examined. If no circular chromosomes are present, a weight of 1 is appended to the list of ‘child weights’ associated with the node. If a circular chromosome is formed, a weight of 0.5 is appended to the list of ‘child weights’ for that operation, as well as for the operation that is to proceed it.

These lists of ‘child weights’ are used during the construction of the network of solutions. When an edge connecting two nodes is added to the graph, it is assigned the weight corresponding to the DCJ operation separating the two nodes.

The above amendments relate to transpositions and block-interchanges. It was thus deemed appropriate to test the newly amended algorithm using two types of test cases. For the first, the source genome would be generated using at least one transposition or block-interchange event. For the second, no transpositions would be applied during source genome generation.

2.3.4.1. Results

The results below show the type of output generated by the Genolve algorithm before and after the amendment (*i.e.*, the alteration of certain edge weights). Results for an example test set falling under two different test cases are shown.

For the first test set, the generation of the source genome from the target genome included a transposition event to determine whether the amendment made to the algorithm did influence the results. For the second test set, not transpositions or block-interchanges were used in the generation of the source genome. For this second test set, no differences between the results before and after the amendments were expected.

Test case 1: with transpositionsSource genome generation:*Target_genome = [[1,2,3,4,5,6,7], [8,9,10,11,12]]*

Rearrangement events applied:

1. Inversion of sequence blocks 2-3:

Intermediate_1 = [[1,-3,-2,4,5,6,7], [8,9,10,11,12]]

2. Transposition of segment 9 to in between sequence blocks 6 and 7:

Intermediate_2 = [[1,-3,-2,4,5,6,9,7], [8,10,11,12]]

3. Fission at sequence blocks 10 and 11:

*Source_genome = [[1,-3,-2,4,5,6,9,7], [8,10], [11,12]]*Results:

	Before amendments	After amendments
Number of shortest paths	30	6
Number of paths containing transpositions	6	6
Total cost per path	4	3

Test case 2: without transpositionsSource genome generation:*Target_genome = [[1,2,3,4], [5,6,7,8,9], [10,11]]*

Rearrangement events applied:

1. Inversion of sequence blocks 2-4:

Intermediate_1 = [[1,-4,-3,-2], [5,6,7,8,9], [10,11]]

2. Balanced translocation of sequence blocks 7-9 and 11:

Intermediate_2 = [[1,-4,-3,-2], [5,6,11], [10,7,8,9]]

3. Inversion of sequence blocks 7-8:

Source_genome = [[1,-4,-3,-2], [5,6,11], [10,-8,-7,9]]

Results:

	Before amendments	After amendments
Number of shortest paths	9	9
Number of paths containing transpositions	0	0
Total cost per path	3	3

2.3.4.2. Discussion

The changes made to the algorithm related to transposition and block-interchange events. Differences in output before and after the algorithm was amended were expected *only* when either of these two events was required to transform one genome into another.

This was the case. For all test sets where transpositions or block-interchanges were utilized to generate the source genome, fewer solutions (shortest paths through the network) were returned by the algorithm *after* the amendments were made.

Furthermore, prior to the algorithm's amendment, the total cost per path/solution was higher than the number of *rearrangements* used in cases where transpositions or block-interchanges were used for genome transformation. Once the algorithm was amended, the total cost per path was an accurate reflection of the number of rearrangements used.

For test sets where transpositions and block-interchanges were absent during source genome generation, no difference in output before and after changes to the algorithm were applied was observed. The number of shortest paths through the network remained constant and, in all cases, the total cost per path corresponded with the number of rearrangements utilized to generate the source genome.

The amendments made thus ensured that correct cost was assigned to transpositions and block-interchanges which in turn (i) decreased the solution space in cases where these types of events were present and (ii) ensured that the total path cost metric accurately reflected the number of rearrangement events that was necessary to transform the source genome into the target genome.

2.3.5. IV: The block-interchange problem

In terms of DCJ operation, the main difference between transpositions and block-interchanges relates to the adjacency formed in the creation of the circular chromosomal intermediate. In the case of transpositions, the same adjacency created during circularization is destroyed in the proceeding linearization step (Fig 18a). In contrast, the adjacency that is destroyed during the linearization step of a block-interchange operation differs from the one created during the previous circularization step (see Fig 18b). The latter allows for the reshuffling of sequence blocks within the circular chromosome prior to its reincorporation (Fig 18b).

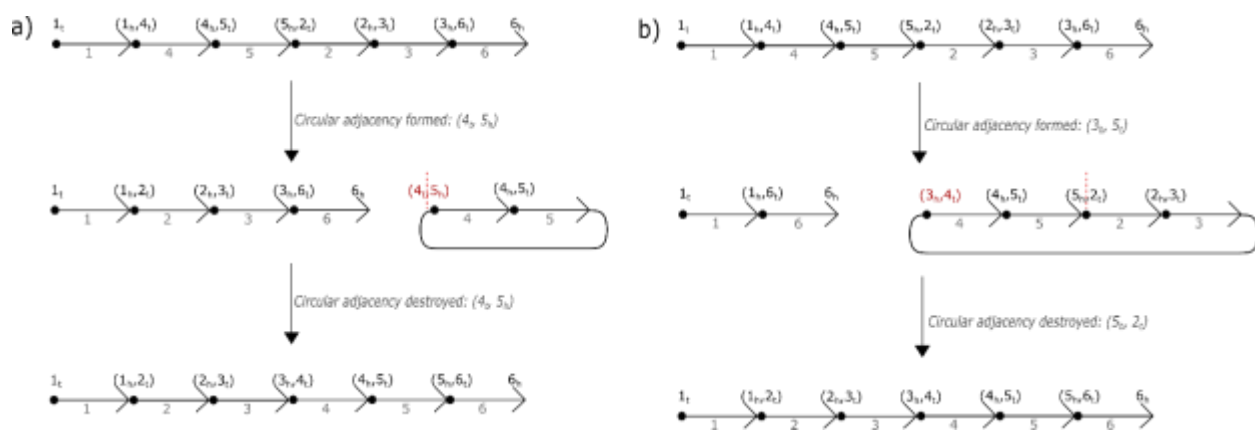


Fig 18. Illustration of the difference between transpositions and block-interchanges using the following source, target genome pair: source genome = $[1, 4, 5, 2, 3, 6]$ and target genome = $[1, 2, 3, 4, 5, 6]$. Arrows represent the genes with the integer label of a gene positioned under the arrow. Telomers and adjacencies are shown at the edges and at the meeting point of two genes, respectively. Red adjacencies represent a newly created adjacency that resulted in the circularization of a segment of sequence blocks. The red dashed lines are used to indicate the adjacency that will be destroyed in order to linearize the circular component. a) Shows a transposition operation used to achieve genome transformation. The adjacency formed during the circularization step is equivalent to the one that is destroyed during the proceeding linearization step. b) Transformation is achieved using a block-interchange event. The adjacency created during circularization differs from the one that is destroyed during the following linearization step.

This reshuffling of sequence blocks or the swapping of two non-adjacent segments of DNA in the genome is not known to occur in nature. Permitting the occurrence of such a rearrangement consequently has a negative impact on the model's biological feasibility.

A solution to this problem would be the prevention of block-interchanges which can be achieved by incorporating the following condition into the algorithm:

The adjacency that is destroyed during the linearization of a circular chromosome must be equivalent to the adjacency created during the previous circularization step.

This condition needed to be incorporated into the rDCJ model itself and was accomplished by adjusting the algorithm as follows:

1. After executing a DCJ operation, the algorithm will evaluate the list of chromosomes of the resulting genome. If the number of circular chromosomes is not zero, the algorithm will store in memory the adjacency that was previously formed **and** is in the list of circular chromosomes.
2. An additional function will be called that uses the stored adjacency to identify a DCJ operation resulting in reincorporation of the circular chromosome.
3. The DCJ operation that was identified will be executed, and the algorithm will proceed as normal.

The exclusion of a certain type of rearrangement event does changes how the rDCJ model functions on an intrinsic level and it therefore becomes necessary to ensure that the model behaves appropriately, i.e. results in complete transformation of the source genome into the target genome despite any changes that are applied to it.

To this end the algorithm was tested with two types of synthetic datasets once the amendments were made.

Source-target genome pairs falling under the first type of dataset were generated by taking a target genome and applying a known series of rearrangements to produce the source genome. Those rearrangements applied to the target genome corresponded with the types of changes made to the model to allow for the evaluation of whether the adjustments made achieved the desired outcome. In this case, numerous transpositions were used during source genome construction to determine whether the algorithm successfully avoids using block-interchanges.

Due to the fact that changes are being made the model itself, however, rigorous testing becomes necessary to ensure that, despite the amendments, the algorithm is still capable of solving *all* source-target genome pairs. The second type of dataset the algorithm was tested with thus included source-target genome pairs where the process of source genome generation was randomized. The

type, location and order of rearrangements applied to a target genome to generate a source genome was chosen at random.

Results for one relevant test set for each of the two types of datasets are shown in the section below.

Results

Example 1: Type 1 Dataset

Results:

	Before amendments	After amendments
Paths returned by the algorithm	Paths 1,2 and 3	Paths 1 and 3
Number of paths	3	2
Number of paths containing block-interchanges	1	0

target_genome = [[1,2,3,4,5,6]]

source_genome = [[1,4,5,2,3,6]]

Path 1: (transposition)

[[1,4,5,2,3,6]] → [[1,2,3,6],[‘o’,4,5]] adjacency resulting in circularization: (4, 5.5)

[[1,2,3,6],[‘o’,4,5]] → [[1,2,3,4,5,6]] adjacency resulting in linearization: (4, 5.5)

Path 2: (block-interchange)

[[1,4,5,2,3,6]] → [[1,6],[‘o’4,5,2,3]] adjacency resulting in circularization: (3.5, 4)

[[1,6],[‘o’4,5,2,3]] → [[1,2,3,4,5,6]] adjacency resulting in linearization: (5.5, 2)

Path 3: (transposition)

[[1,4,5,2,3,6]] → [[1,4,5,6],[‘o’,2,3]] adjacency resulting in circularization: (2, 3.5)

[[1,4,5,6],[‘o’,2,3]] → [[1,2,3,4,5,6]] adjacency resulting in linearization: (2, 3.5)

Path 1 and 2 in Example 1

Input

target_genome = [[1,2,3,4,5,6]]

source_genome = [[1,4,5,2,3,6]]

Generate adjacency lists

target_adjacencies = [1, 6.5, (1.5, 2), (2.5, 3), (3.5, 4), (4.5, 5), (5.5, 6)]

source_adjacencies = [1, 6.5, (1.5, 4), (4.5, 5), (5.5, 2), (2.5, 3), (3.5, 6)]

#1. Find elements present in target_adjacencies but not source adjacencies

The first element present in target_adjacencies but not in source_adjacencies is (1.5, 2)

The second element present in target_adjacencies but not in source_adjacencies is (3.5, 4)

Path 1:

element = (1.5, 2)

Path 2:

element = (3.5, 4)

Path 1

2. Execute the required DCJ operation to create the element

The DCJ operation to create (3.5, 4) involves the following two adjacencies in the intermediary_genome: (, 4) and (2, 5.5).

adjacency_1 = (1.5, 4)

adjacency_2 = (2, 5.5)

new_adjacency_1 = (1.5, 2)

new_adjacency_2 = (4, 5.5)

adjacencies_destroyed = (*adjacency_1*, *adjacency_2*)

#these adjacencies are removed from the adjacency list
 adjacencies_formed = (new_adjacency_1, new_adjacency_2)
these adjacencies are added to the adjacency list

#use adjacency list to calculate new genome
 intermediary_genome = [[1,2,3,6,],[‘o’,4,5]]

3. Was a circular chromosome formed?

Yes.

circularization_adjacency = (4, 5.5)

3.1 Find the partner of the circularization_adjacency in the set of linear chromosomes

The final-state adjacency in which the extremity 4 occurs is (3.5, 4). In the intermediary genome, the extremity 3.5 occurs in the adjacency (3.5, 6) – this adjacency is present on a linear chromosome making it a viable linearization partner for circular adjacency (4, 5.5)

final_state_adjacency = (3.5, 4)

linearization_partner = (3.5, 6)

3.2. Execute the required DCJ for linearization and reinsertion

circularization_adjacency = (4, 5.5)

linearization_partner = (3.5, 6)

final_state_adjacency = (3.5, 4)

new_adjacency_2 = (5.5, 6)

adjacencies_destroyed = (adjacency_1, adjacency_2)

#these adjacencies are removed from the adjacency list

Path 2

#use adjacency list to calculate new genome

intermediary_genome = [[1,6,],['o',4,5,2,3]]

3. Was a circular chromosome formed?

Yes.

circularization_adjacency = (3.5, 4)

3.1 Find the partner of the circularization_adjacency in the set of linear chromosomes

The final-state adjacency in which the extremity 3.5 occurs is (3.5, 4). In the source genome, the extremity 3.5 occurs in the adjacency (3.5, 4) – thus it is already present in its final-state adjacency. Therefore, there is no DCJ operation possible for the linearization and reinsertion of the circular chromosome and the algorithm is unable to continue with/complete the transformation process.

Thus:

→ move to the next path

2. Execute the required DCJ operation to create the element

The DCJ operation to create (3.5, 4) involves the following two adjacencies in the intermediary_genome: (1.5, 4) and (3.5, 6).

adjacency_1 = (1.5, 4)

adjacency_2 = (3.5, 6)

new_adjacency_1 = (3.5, 4)

new_adjacency_2 = (1.5, 6)

adjacencies_destroyed = (adjacency_1, adjacency_2)

#these adjacencies are removed from the adjacency list

adjacencies_formed = (new_adjacency_1, new_adjacency_2)

these adjacencies are added to the adjacency list

Example 2: Type 2 Dataset

Input

```
target_genome = [[1,2,3,4,5,6,7,8]]
source_genome = [[1,6,7,4,5,2,3,8]]
```

Generate adjacency lists

```
target_adjacencies = [1, 8.5, (1.5, 2), (2.5, 3), (3.5, 4), (4.5, 5), (5.5, 6), (6.5, 7), (7.5, 8)]
source_adjacencies = [1, 8.5, (1.5, 6), (6.5, 7), (7.5, 4), (4.5, 5), (5.5, 2), (2.5, 3), (3.5, 8)]
```

#1. Find elements present in target_adjacencies but not source_adjacencies

```
# The first element present in target_adjacencies but not in source_adjacencies is (1.5, 2)
element = (1.5, 2)
```

2. Execute the required DCJ operation to create the element

```
# The DCJ operation to create (1.5, 2) involves the following two adjacencies in the intermediary_genome: (1.5, 6) and (2, 5.5).
```

```
adjacency_1 = (1.5, 6)
adjacency_2 = (2, 5.5)
new_adjacency_1 = (1.5, 2)
new_adjacency_2 = (5.5, 6)
```

```
adjacencies_destroyed = (adjacency_1, adjacency_2)
#these adjacencies are removed from the adjacency list
```

```
adjacencies_formed = (new_adjacency_1, new_adjacency_2)
# these adjacencies are added to the adjacency list
```

```
#use adjacency list to calculate new genome
intermediary_genome = [[1,2,3,8,],[ 'o',6,7,4,5]]
```

3. Was a circular chromosome formed?

Yes.

circularization_adjacency = (5.5, 6)

3. 1 Find the partner of the circularization_adjacency in the set of linear chromosomes

The final-state adjacency in which the extremity 5.5 occurs is (5.5, 6). In the source genome, the extremity 5.5 occurs in the adjacency (5.5 6) – thus, it is already present in its final-state adjacency. Therefore, there is no DCJ operation possible for the linearization and reinsertion of the circular chromosome, and the algorithm is unable to continue with/complete the transformation process.

Thus:

→ move to the next path

However, in this example, if the same steps outlined above are followed for each element present in the target_adjacencies but not in the source_adjacencies, it will become apparent that all the paths terminate in the same way – where no DCJ operation for linearization and reinsertion exists. Thus, the algorithm is unable to find even a single solution for the transformation of the source genome into the target genome.

2.3.5.1. Discussion

When testing the algorithm on Type 1 Datasets, it generated the desired outcome in all cases. The strategy used and amendments made to the algorithm were successful. The restriction placed on the adjacency to be used during linearization excluded the execution of block-interchange events, ensuring that only biological events are permitted during the transformation of the source genome into the target genome.

A problematic and unforeseen outcome was however encountered twice while testing the algorithm on Type 2 datasets. The algorithm was unable to solve/complete the sorting process of

the source genome into the target genome, i.e. not a single path from source to target was identifiable. Upon closer examination of these scenarios, it became apparent that the problem occurred due to the inherent nature of the rDCJ model.

If the adjacency created during circularization is a final-state adjacency, no DCJ operation that destroys it will exist, and the path cannot be completed, *i.e.* transformation into the target genome remains incomplete. This does not present a problem as long as at least one other path, which does not terminate at such an adjacency, exists. In cases where no other paths exist, however, the algorithm is unable to sort the source genome into the target genome successfully.

As long as those rearrangements applied to the target genome during source genome generation are possible from an evolutionary perspective, the algorithm needs to be able to transform the source genome back into the target genome.

The restriction placed on the algorithm permitting only the destruction of those circular adjacencies that result in the formation of a circular chromosome thus has a negative effect on the utility of the algorithm as it may lead to unsolvable scenarios.

Consequently, it became apparent that, to utilize the rDCJ model, block-interchange events have to be permitted. This was considered to be problematic because block-interchanges remained unfeasible on a biological level. A different strategy to negate the negative effect that allowing this type of rearrangement has on biological accuracy, whilst maintaining the utility of the model, thus needed to be devised.

The development of the new strategy and the corresponding results and discussion, follow.

2.3.5.2. *Method*

The conclusion of the discussion above is that block-interchange events have to be permitted. Upon closer inspection, this type of event can, in essence, be viewed as two independent transposition operations. For the second, ‘unsolvable’ example given in the results above, the two transposition events would be those shown in Fig 19.

If we accept that a block-interchange event represents two separate transposition events, the algorithm can be adjusted accordingly.

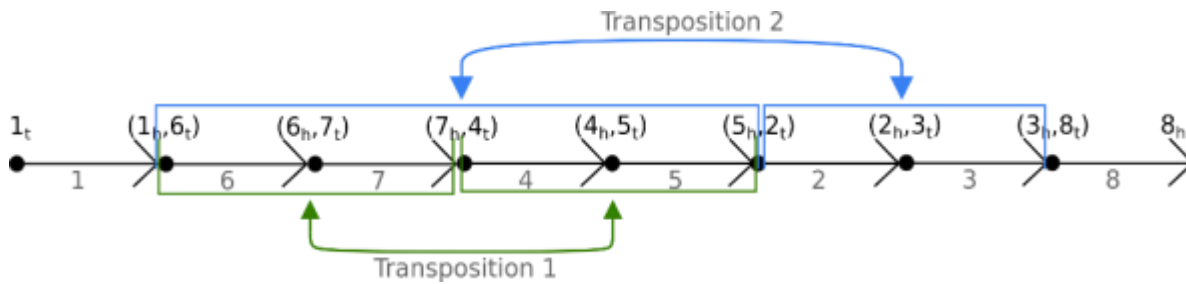


Fig 19. Illustration of the two transposition events that constitute a single block-interchange event. The block-interchange event is comprised of the swapping of sequence blocks 2 and 3 with sequence blocks 6 and 7. This event involves the transposition of sequence blocks 6 and 7 to in between sequence block 5 and 2 (shown in green) and the transposition of sequence blocks 6, 7, 4 and 5 to in between sequence blocks 3 and 8 (shown in blue).

In solving the ‘operation cost’ problem, the DCJ operations required for the execution of a transposition or block-interchange operation were each assigned a cost of 0.5 so that the total cost of the rearrangement event would equate to one. If a block-interchange event represents two successive transposition events, the total cost of this rearrangement should be 2.

The algorithm was amended to achieve this as follows:

1. After executing a DCJ operation, the algorithm will evaluate the list of chromosomes of the resulting genome. If the number of circular chromosomes is not zero, the algorithm will store in memory the adjacency that was previously formed **and** is in the list of circular chromosomes. This DCJ operation will be assigned a cost of 0.5.
2. The adjacency that is destroyed in the following linearization operation will be compared to the one that was previously stored in memory. If the two adjacencies are equivalent, the linearization and reinsertion DCJ operation will be assigned a cost of 0.5 else it will be assigned a cost of 1.5.

The algorithm was once again tested with datasets of type 1 and 2. The amendments ensured that the correct cost was assigned to block-interchanges, whilst allowing the algorithm to transform all source genomes into their corresponding target genomes.

By viewing block-interchange events as two successive transpositions, it is possible to utilize the rDCJ model while negating the negative impact permitting the block-interchange event may have on the model’s biological accuracy.

2.3.6. V: From DCJ operations to genome rearrangements

The rDCJ model utilizes and reports on a set of DCJ operations required to transform a source genome into a target genome. To study the process of structural genomic rearrangements it becomes necessary to translate these DCJ operations into the genomic rearrangement events they represent. To this end, an additional class of functions for the identification of rearrangement events was created.

This class of functions assigned a rearrangement identity to a DCJ operation based on:

- (i) the combination and location (interchromosomal vs intrachromosomal operation) of adjacencies and extremities that are being destroyed,
- (ii) the combination and location of adjacencies and extremities that are being formed and
- (iii) the type of chromosomes (linear or circular) resulting from the DCJ operation.

The algorithm was tested with datasets of Type 1, and its capability of identifying each of the different types of rearrangement events was confirmed.

2.3.7. VI: Incorporating the expected frequency of occurrence per rearrangement via a weighting system

Certain types of rearrangements have a higher frequency of occurrence in some species and are thus more likely to transpire during genomic evolution. Therefore, it was deemed appropriate to incorporate an adjustable set of relative rearrangement frequencies that would alter the weights of the different types of rearrangements. These weights would be assigned to the edges in the network of solutions, thereby influencing the total weight or cost of the various ‘paths’ from the source genome to the target genome.

As a result of solving the operation cost problem at this stage of development, a weighting system was already in place. DCJ operations that did not entail the formation or destruction of circular chromosomes were assigned a weight of one whilst those that did were assigned a weight of 0.5 or 1.5 depending on whether formed part of a transposition or block-interchange event.

In order to adjust these weights to reflect the relative frequencies in which the rearrangement events were expected to occur it would, however, become necessary to identify the types of

rearrangement events represented by each DCJ operation. This had been accomplished in the previous phase of development.

Allowing user-adjustable, relative expected frequencies in which rearrangements occur thus required that:

1. A method with which to translate frequencies into edge weights be developed
2. The necessary amendments to the algorithm are made to incorporate these calculations

The user input would be in the form of a ratio of the number of times each type of rearrangement is expected to occur relative to every other rearrangement type. For example, one may expect inversions to occur twice as frequently as any other rearrangement giving you a ratio of 2 inversions: 1 of all other rearrangement types. Alternatively, for every one chromosome fusion event, you may expect one chromosome fission, one unbalanced translocation, two balanced translocations, three transpositions and four inversions. The input ratios will then be 1 fusion: 1 fission: 1 unbalanced translocation: 2 balanced translocations: 3 transpositions: 4 inversions.

The following calculation was used to transform these relative frequencies into a set of rearrangement weights:

Given the list of ratios $R = [a, b, c, d, e, f]$, where a-f are integers representing the relative frequency of occurrence of the different rearrangement events:

$\text{max} = \text{maximum value in } R$

For each element in R:

If the element = 0:

Replace the element with $(\text{max} \times 10)$

Else:

Replace the element with $\frac{\text{max}}{\text{element}}$

If the expected relative frequency of occurrence is non-zero, the rearrangement is assigned a weight that equates to the value of the rearrangement with the highest relative expected frequency,

divided by the rearrangement's relative frequency value. This ensures that the weight of those rearrangements of which the occurrence is most likely is always equal to one and all other rearrangement weights are scaled accordingly.

If an element is expected to occur 0% of the time, it is necessary to weight it heavily enough that it appears in the output only if no other rearrangement or combination of rearrangements can replace it. For this reason, rearrangements with a relative expected frequency of occurrence of zero are given a weight ten times that of the maximum relative frequency value.

The weighting system already in place, assigned a weight of 0.5 or 1.5 to the DCJ operations representing transpositions and block-interchanges and a weight of 1 to DCJ operations representing all other rearrangements. These values were multiplied by those in the newly calculated list of rearrangement weights, ensuring that the correct cost of transpositions and block-interchanges relative to other rearrangements were maintained whilst incorporating the additional weighting system.

The final rearrangement weight values were then assigned to the relevant edges in the network of solutions. Those edges representing high-frequency rearrangements will have lower weights. These differently weighted edges effect which 'paths' of genome transformation are included in the output as the algorithm finds and returns all the 'lowest cost' paths (paths with the lowest cumulative edge weight) through the network of solutions. The higher the number of low weight rearrangements within a path, the higher the likelihood that the path is included in the output.

The Fig 20 bellow illustrates how different expected rearrangement relative frequency of occurrence ratios affects the paths/solution generated as output by the algorithm.

Given the source genome $[[1,-3,2],[4],[5,7,6]]$ and target genome $[[1,2,3],[4,5],[6,7]]$ the sorting process of the former into the latter can be divided into two independent parts – the first part sorts chromosome 1 and the second, chromosomes 2 and 3 (Fig 20a). The sorting process in both parts can be achieved either with a single transposition or alternatively with two inversions in the case of part one and two unbalanced translocations in the case of part two (Fig 20a).

The symbol for each DCJ operation is shown in bold italic next to the operation in Fig 20a. These symbols correspond with those labeling their corresponding edges in the network of solutions shown in Fig 20b.

The integers in the blocks at the bottom of each solution or path of transformation (Fig 20b) is simply used to identify the path. For example, Path 1 and Path 2 consist of the following ordered sets of DCJ operations, respectively: {I1, I2, T2a, T2b} and {I1, I2, U1, U2}.

There are 14 possible paths because the sorting of chromosome 1 and the sorting of chromosomes 2 and 3 are independent processes and can thus occur in any order. Inversion 1 (I1) and Inversion 2 (I2) are not independent (i.e. I1 must precede I2) however I2 need not occur directly after I1 (i.e. they can be separated by some other rearrangement event) – the same is true for Unbalanced translocation 1 (U1) and 2 (U2).

This is not, however, the case for Transposition 1a and 1b or 2a and 2b where b must proceed a *directly* (see the restricted DCJ model in Chapter 2).

The table in Fig 20c shows which paths would appear in the algorithm's output under different relative expected frequency of occurrence ratios. If all rearrangements are expected to occur with equal frequencies (a) then each edge representing a DCJ operation forming part of a transposition will have a weight of 0.5 whilst those representing inversions and unbalanced translocation will have weights of 1. This makes a single transposition 'cheaper' than two inversion events or two unbalanced translocations. The algorithm will thus return those paths consisting only of transpositions (Paths 6 and 8).

In Fig 20c (b) the ratio shows that inversions are expected to occur twice as frequently as all other types of rearrangements. By applying the calculation outlined above, the weights of the different DCJ operations equates to:

Transposition DCJ operations: $T1a = T1b = T2a = T2b = 0.5 \times \frac{2}{1} = 1$

Inversion DCJ operations: $I1 = I2 = 1 \times \frac{2}{2} = 1$

Unbalanced translocation DCJ operations: $U1 = U2 = 1 \times \frac{2}{1} = 2$

The total cost of two inversions thus becomes the same as the cost of the two DCJ operations making up a transposition and either can be used to sort chromosome one. Two unbalanced

translocations remain more costly than one transposition. In addition to Paths 6 and 8 the algorithm will now also include in its output Paths 1, 5 and 9.

If we apply the same logic and calculation to the scenario in Fig 20c (c), the weights of the DCJ operations become:

Transposition DCJ operations: $T1a = T1b = T2a = T2b = 0.5 \times \frac{4}{1} = 2$

Inversion DCJ operations: $I1 = I2 = 1 \times \frac{4}{4} = 1$

Unbalanced translocation DCJ operations: $U1 = U2 = 1 \times \frac{4}{1} = 4$

The cost of two inversions is now lower than that of a single transposition (whilst one transposition remains cheaper than two unbalanced translocations). The paths included in the output will therefore include only those that use inversions to sort chromosome one and a transposition to sort chromosomes two and three.

2.3.8. The output of Genolve and how to interpret it

The rest of this thesis contains numerous figures depicting the output generated by Genolve for different input genome pairs. It is thus important to give a comprehensive outline of the format of the output Genolve generates and instruction on how the output should be interpreted.

In this section the output of Genolve generated for the following example input genome pair: source genome = [10, 1, -2, 3, 5, -4, 9], [7], [8, 6], [11], target genome = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]] is shown (Fig 21) and discussed in detail.

An output summary is generated by Genolve which includes the source and target genomes, the number of solutions found by Genolve ('Number of most parsimonious solutions'), the average number of operations per solution and the average number of each individual type of operation per solution. In the output for the example above Genolve identified 720 solutions with six operations per solutions, where the types of operations included within a solution generally consisted of one inversion, one transposition, one balanced translocation, one unbalanced translocation, one fission, one fusion and zero block-interchanges.

Following the results summary, each of the individual solutions is listed. In Fig 21, only the first three solutions are shown, the results file does, however, contain each of the 720 solutions.

a) **target_genome** = $[[1,2,3],[4,5],[6,7]]$
source_genome = $[[1,-3,2],[4],[6,5,7]]$

Part 1: sorting $[1,-3,2]$ into $[1,2,3]$ Part 2: sorting $[4][6,5,7]$ into $[4,5][6,7]$

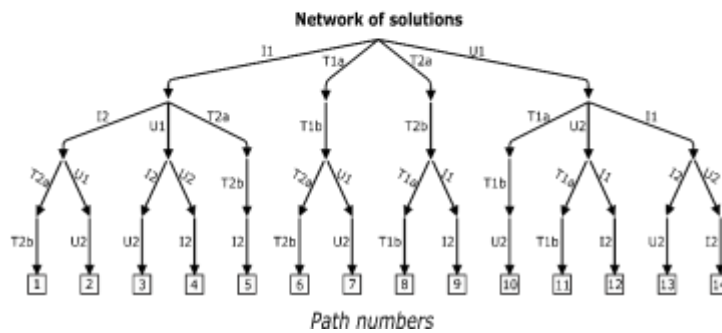
(i) 1 Transposition:
 $[1,-3,2] \rightarrow [1,2][\text{'o'},3]$ **T1a**
 $[1,2][\text{'o'},3] \rightarrow [1,2,3]$ **T1b**

(ii) 2 Inversions:
 $[1,-3,2] \rightarrow [1,-3,-2]$ **I1**
 $[1,-3,-2] \rightarrow [1,2,3]$ **I2**

(i) 1 Transposition:
 $[4][6,5,7] \rightarrow [4][6,7][\text{'o'},5]$ **T2a**
 $[4][6,7][\text{'o'},5] \rightarrow [4,5][6,7]$ **T2b**

(ii) 2 Unbalanced translocations:
 $[4][6,5,7] \rightarrow [4,5,7][6]$ **U1**
 $[4,5,7][6] \rightarrow [4,5][6,7]$ **U2**

b)



c)

Ratios of expected relative rearrangement frequency*	Paths present in output
a) 1:1:1:1:1:1	6 & 8
Changing inversion frequency	
b) 2:1:1:1:1:1	1, 5, 6, 8 & 9
c) 4:1:1:1:1:1	1, 5 & 9
Changing unbalanced translocation frequency	
d) 1:1:1:2:1:1	6, 7, 9, 10 & 11
e) 1:1:1:4:1:1	7, 10 & 11
Changing inversion and unbalanced translocation frequencies	
f) 2:1:1:2:1:1	all paths
g) 4:1:1:4:1:1	2, 3, 4, 12, 13 & 14

* Ratios are given as inversion:transpositions:balanced translocations:unbalanced translocations:fissions:fusions

Fig 20. a) Depiction of how the transformation of the source genome $[[1,-3,2],[4],[6,5,7]]$ into the target genome $[[1,2,3],[4,5],[6,7]]$ can be divided into two independent parts involving sequence blocks 1, 2 and 3 and sequence blocks 4, 5, 6 and 7 respectively. Part one, the sorting of the chromosome $[1,-3,2]$ into $[1,2,3]$ can be achieved using either (i) one transposition event that compromises a circularization (T1a) followed directly by a linearization (T1b) operation or by using (ii) two inversions (I1 and I2). The second part achieving the transformation of chromosomes $[4],[6,5,7]$ into $[4,5],[6,7]$ is obtainable either by application of a transposition event constituting a circularization (T2a) and circularization (T2b) operation or by the use of two unbalanced translocations (U1 and U2). b) The network of solutions showing all possible combinations of the different rearrangement event that can be used to successfully transform the source into the target genome. There are a total of 14 different solution or paths through the network (shown at the bottom). c) a table showing different weighting ratios that can be applied to the network (left) and the path that would be included in the output of an algorithm finding the lowest cost/weight paths through the network (right).

```

#####

Source Genome: [[10, 1, -2, 3, 5, -4, 9], [7], [8, 6], [11]]
Target Genome: [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]

Number of most parsimonious solutions: 720

Average number of operations per solution: 6.0

Average number of each operation per solution:
Inversions: 1.0 Transpositions type 1: 1.0 Balanced translocations: 1.0
Unbalanced translocations: 1.0 Fusions: 1.0 Fissions: 1.0 Block-interchanges: 0.0

Solutions:

Solution number 1
[[[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], [11]], ('none, this is the source genome', 'N/A')]
[[[-6, -5, -3, 2, -1, -10], [7], [8, -4, 9], [11]], ('b_trl', (((4.5, 5.5), (6, 8.5)), ((4.5, 8.5), (5.5, 6)))))]
[[[-6, -5, -3, 2, -1, -10], [7], [8, 9], [11], ['o', 4]], ('trp0', (((4, 9), (4.5, 8.5)), ((4, 4.5), (8.5, 9)))))]
[[[-6, -5, -4, -3, 2, -1, -10], [7], [8, 9], [11]], ('trp1', (((3.5, 5), (4, 4.5)), ((3.5, 4), (4.5, 5)))))]
[[[-3, 2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('fis', ((3.5, 4), (3.5, 4)))]
[[[-3, -2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))]
[[[-3, -2, -1, -10], [4, 5, 6], [7, 8, 9], [11]], ('fus', (7.5, 8, (7.5, 8)))))]
[[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]], ('u_trl', (((1, 10.5), 11), ((10.5, 11), 1)))))]

Solution number 2
[[[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], [11]], ('none, this is the source genome', 'N/A')]
[[[-6, 4, -5, -3, 2, -1, -10], [7], [8, 9], [11]], ('b_trl', (((4, 9), (6, 8.5)), ((4, 6), (8.5, 9)))))]
[[[-6, -5, -3, 2, -1, -10], [7], [8, 9], [11], ['o', 4]], ('trp0', (((4, 6), (4.5, 5.5)), ((4, 4.5), (5.5, 6)))))]
[[[-6, -5, -4, -3, 2, -1, -10], [7], [8, 9], [11]], ('trp1', (((3.5, 5), (4, 4.5)), ((3.5, 4), (4.5, 5)))))]
[[[-3, 2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('fis', ((3.5, 4), (3.5, 4)))]
[[[-3, -2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))]
[[[-3, -2, -1, -10], [4, 5, 6], [7, 8, 9], [11]], ('fus', (7.5, 8, (7.5, 8)))))]
[[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]], ('u_trl', (((1, 10.5), 11), ((10.5, 11), 1)))))]

Solution number 3
[[[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], [11]], ('none, this is the source genome', 'N/A')]
[[[-3, 2, -1, -10], [5, -4, 9], [-6, -8], [7], [11]], ('fis', ((3.5, 5), (3.5, 5)))]
[[[-3, 2, -1, -10], [5, 6], [7], [8, -4, 9], [11]], ('b_trl', (((4.5, 5.5), (6, 8.5)), ((4.5, 8.5), (5.5, 6)))))]
[[[-3, 2, -1, -10], [5, 6], [7], [8, 9], [11], ['o', 4]], ('trp0', (((4, 9), (4.5, 8.5)), ((4, 4.5), (8.5, 9)))))]
[[[-3, 2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('trp1', (((4, 4.5), 5), ((4.5, 5), 4)))))]
[[[-3, -2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))]
[[[-3, -2, -1, -10], [4, 5, 6], [7, 8, 9], [11]], ('fus', (7.5, 8, (7.5, 8)))))]
[[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]], ('u_trl', (((1, 10.5), 11), ((10.5, 11), 1)))))]

...

#####

```

Fig 21. An example of the output generated by the Genolve tool. The above output shows the results for the input genome pair: source genome = [[10, 1, -2, 3, 5, -4, 9], [7], [8, 6], [11]] and target genome = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]. The '...' at the bottom of the figure indicates that only three of the 720 solutions generated by Genolve are shown in the figure. All solutions are present in the output file, however. Each line within a solution contains (a genome, (the type of rearrangement that resulted in the genome, ((the adjacencies/telomeric ends that were destroyed/'cut' during the rearrangement), (the adjacencies/telomeric ends that were newly created by the rearrangement))), with x denoting the tail extremity and $x.5$ the head extremity of gene x . A genome is contained within a set of square brackets and each individual chromosome making up a genome is also contained within a set of square brackets. If the first index of a chromosome is 'o', it indicates the circular nature of that chromosome. The following abbreviations for the different type of rearrangements are used: inversion: inv, circularization operation of a transposition/block-interchange: trp0, reinsertion operation of a transposition: trp1, reinsertion operation of a block-interchange: trp2, balanced translocation: b_trl, unbalanced translocation: u_trl, chromosome fission: fis, chromosome fusion: fus.

An individual solution can be read as follows:

Solution number: *number of the specific solution within the solution set*

Each of the lines within a solution is in the following format:

(a genome, (the type of rearrangement that resulted in the genome, (the adjacencies/telomeric ends that were destroyed/'cut' during the rearrangement), (the adjacencies/telomeric ends that were newly created by the rearrangement))).

The different types of rearrangements are abbreviated as follows: inversion: *inv*, circularization operation of a transposition/block-interchange: *trp0*, reinsertion operation of a transposition: *trp1*, reinsertion operation of a block-interchange: *trp2*, balanced translocation: *b_trl*, unbalanced translocation: *u_trl*, chromosome fission: *fis*, chromosome fusion: *fus*.

Note that the genome in the first line of every solution will be the source genome. There is thus no rearrangement that created it as it is the starting point. There will also be no corresponding destroyed and newly created adjacencies/telomeric ends.

To give a comprehensive illustration of how the output of Genolve should be read I will step through each line in Solution 1 individually. The genome graphs for each of the genomes present in Solution 1, shown in Fig 22, give a visual representation of the intermediary genome at various transformation steps.

Line 1: ([[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], [11]], ('none, this is the source genome', 'N/A'))

Line one contains the source genome and no rearrangement was applied to create it.

Line 2: ([[-6, -5, -3, 2, -1, -10], [7], [8, -4, 9], [11]], ('b_trl', (((4.5, 5.5), (6, 8.5)), ((4.5, 8.5), (5.5, 6))))))

The intermediary genome $[[[-6, -5, -3, 2, -1, -10], [7], [8, -4, 9], [11]]]$ was created by applying a balanced translocation to the source genome. The adjacencies that were destroyed during this rearrangement event was (4.5, 5.5) and (6, 8.5). In other words, a 'cut' was made between sequence blocks [4] and [-5] as well as between [-6] and [-8] in the source genome. The new adjacencies that were created were (4.5, 8.5) and (5.5, 6). In other words, in the intermediary genome, sequence blocks [4] and [-8] as well as [-6] and [-5] are now adjacent. Note that the chromosome [8, -4, 9] in the intermediary genome is equivalent to the chromosome [-9, 4, -8] (it merely depends on the direction in which you read the blocks).

Line 3: ([[-6, -5, -3, 2, -1, -10], [7], [8, 9], [11], ['o', 4]], ('trp0', (((4, 9), (4.5, 8.5)), ((4, 4.5), (8.5, 9)))))

A circularization operation that forms the first part of a transposition or block-interchange event results in the formation of a genome containing the circular chromosome ['o', 4]. The 'o' at the first index of any chromosome is indicative of the circular nature of that chromosome. The genome was generated by cutting the adjacencies (4, 9) and (4.5, 8) between sequence blocks [-4] and [9] and sequence blocks [8] and [-4], respectively. The newly formed adjacencies (4, 4.5) and (8.5, 9) resulted in the formation of the circular chromosome ['o', 4] and the adjacency between sequence blocks [8] and [9] respectively.

Line 4: ([[-6, -5, -4, -3, 2, -1, -10], [7], [8, 9], [11]], ('trp1', (((3.5, 5), (4, 4.5)), ((3.5, 4), (4.5, 5)))))

The reinsertion component of a transposition event resulted in this genome. The adjacencies (3.5, 5) and (4, 4.5) were cut to create new adjacencies between the sequence blocks [-4] and [-3] and the sequence blocks [-5] and [-4], respectively.

Line 5: ([[-3, 2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('fis', ((3.5, 4), 3.5, 4)))

With the event that resulted in this intermediary genome, a fission event, the notation of telomeric ends are encountered. The fission event cuts the adjacency (3.5, 4) in order to create the telomeric ends 3.5 and 4. Note that unlike in the case of adjacencies, telomeric ends, are not contained within their own set of round brackets in Genolve's output.

Line 6: ([[-3, -2, -1, -10], [4, 5, 6], [7], [8, 9], [11]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))

In the sixth line, the intermediary genome was generated by an inversion event that entailed the destruction of adjacencies (1.5, 2.5) and (2, 3) and the created of the adjacencies (1.5, 2) and (2.5, 3).

Line 7: ([[-3, -2, -1, -10], [4, 5, 6], [7, 8, 9], [11]], ('fus', (7.5, 8, (7.5, 8))))

Here a fusion event is encountered. The two telomeric ends 7.5 and 8 are joined together to create the adjacency (7.5, 8). Note again that the telomeric ends are not contained within their own pair of round brackets as adjacencies are.

Line 8: $([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]], ('u_trl', (((1, 10.5), 11), ((10.5, 11), 1))))$

In the final line of the solution the target genome, $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]$, is generated by the application of an unbalanced translocation. This rearrangement involved the destruction of the adjacency $(1, 10.5)$ and the telomeric end 11 to allow for the creating the adjacency $(10.5, 11)$ and the telomeric end 1.

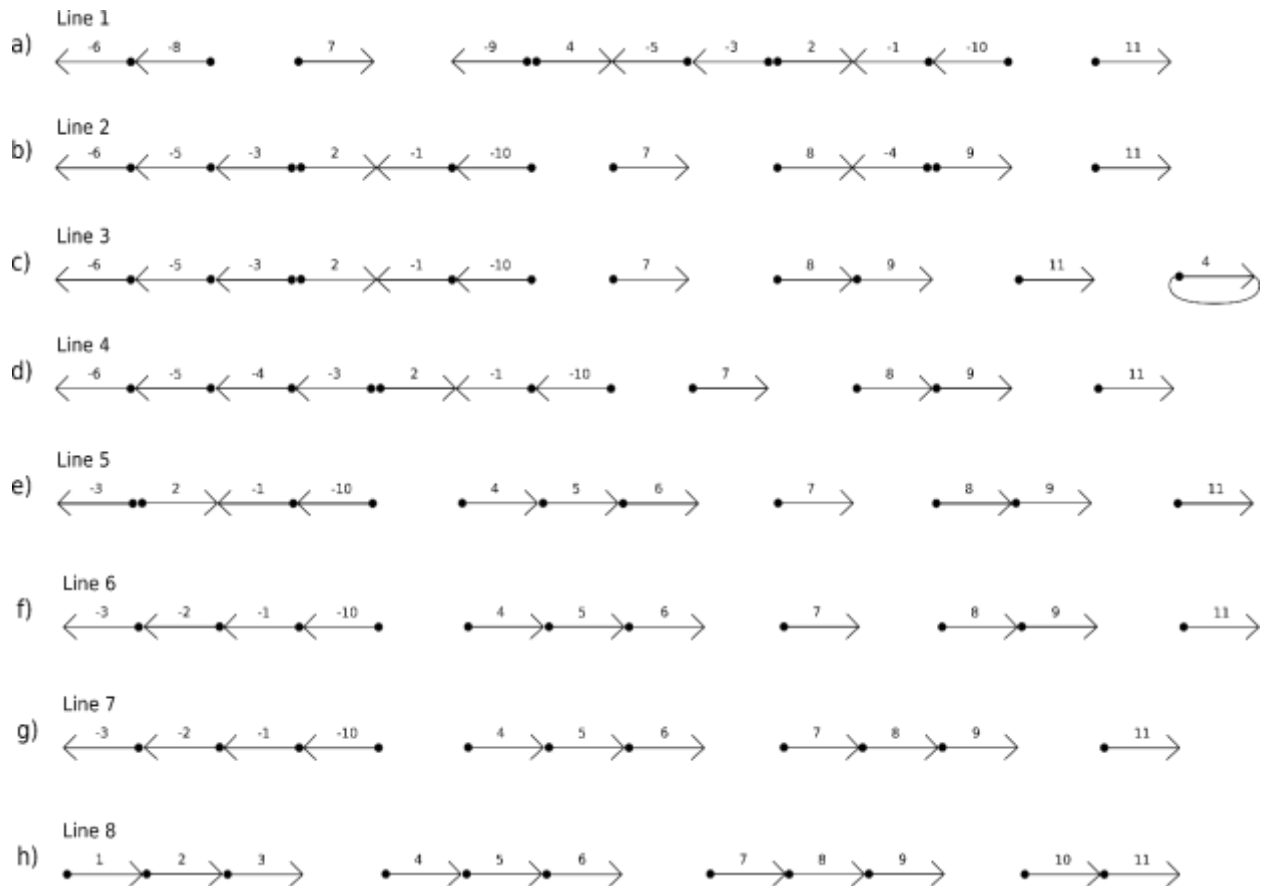


Fig 22. Genome graphs giving a visual representation of the source, target and intermediary genomes present in the different lines of Solution 1 present of the solution set generated by Genolve for the source genome: $[[-6, -8], [7], [-9, 4, -5, -3, 2, -1, -10], 11]$ and target genome: $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11]]$. a) – h) shows the various genomes present in lines 1 – 8 of Solution 1, respectively. Each arrow represents a gene, with the gene label appearing above the arrow. Genes occur in the tail-to-head direction (the tail of the arrow precedes the arrowhead) unless the gene's orientation in the genome is inverted, in which case the head of the arrow representing the gene will precede the tail of the arrow. The inverse orientation of a gene within the genome is also indicated by the negative sign in front of a gene label on top of an arrow. The output generated by Genolve for this source-target genome pair is shown in Fig 21.

Chapter 3: Evaluation of Genolve using synthetic data

Following the Genolve tool's development, it is necessary to evaluate its capabilities, functionality and limitations. Rigorous testing of any tool requires large and diverse datasets. For this purpose, an additional program, christened the Evolver, was developed and used to generate vast amounts of synthetic data.

The most important factor that needed to be confirmed was the Genolve's ability to do what it was developed to do, namely successfully identify the path to transform a source genome into a target genome, for any source-target input genome pair (given that the two input genomes consist of the same set of sequence blocks). This was investigated by running Genolve on a very large number of diverse input genomes and observing whether it is able to generate a solution set for each.

Another factor investigated was the impact of incorporating a correct ratio of expected frequency of occurrence for the different types of rearrangements had on Genolve's ability to identify the true rearrangement scenario that separates a source and target genome. (See Section 2.3.7.). Running Genolve on the same input genome pair, but under different weighting ratios, allowed for the comparison between the percentage of times the output that was generated included the correct rearrangement scenario.

The scalability of the tool was also investigated. Larger, more complex genomes will result in larger solution sets and longer individual solution paths, both of which increase the runtime and memory requirements of the tool. Analyses were thus conducted to determine the relationship between increases in the complexity of genomes and (i) the number of solutions within a solution set, (ii) the average number of rearrangements a single solution consists of and (iii) the average time it takes Genolve to generate the solution set. The memory requirements of Genolve were also investigated.

Chapter three will be organized as follows: the first section will explain how the synthetic datasets were developed, and the notation used to illustrate genomic complexity. The sections that follow will address Genolve's ability to solve all source-target genome input pairs, the effect weight ratios have on the output of Genolve, factors contributing to the scalability of the tool and finally the runtime and memory requirements of the tool, respectively.

3.1 Development of synthetic genomes and genomic complexity notation

3.1.1 Development of synthetic genomes

To evaluate the tool's capabilities and behavior, it is necessary to test it on a large number of source-target genome pairs to ensure that any results are not observed simply due to change as a result of small sample size.

Ideally, these datasets would comprise biological data. Comprehensive evaluation of the tools' behavior and capabilities require a large amount of diverse source-target genome pairs of which the evolutionary events that resulted in the source genome's evolution into the target genome is known. The amount of biological data of this type available was too small to conduct any form of rigorous testing. Also, a synthetic dataset allows us to test positions, types, and frequency of all possible structural events, verifying the tool's accuracy under a set of conditions not necessarily available with biological data.

It was thus necessary to develop an additional program with which to simulate genomic evolution. Such a program would *start* with a target genome and apply a series of rearrangements to generate the source genome. The inverse of this series of rearrangements would then describe the evolutionary events that resulted in the 'evolution' of the source genome into the target genome (Fig 23). This inverse series of rearrangements is called the 'true' evolutionary scenario and represents the events that resulted in the evolution of the source genome into the target genome.

Fig 23a shows how the program, dubbed the Evolver, takes a target genome and applies a set of four rearrangements to it (an inversion, fusion, transposition and balanced translocation in that order) to generate the source genome. As it generates the source genome from the target genome, it records the rearrangement events it applies. Once source genome generation is complete, the Evolver generates a series of rearrangements which is the inverse of those used for source genome transformation

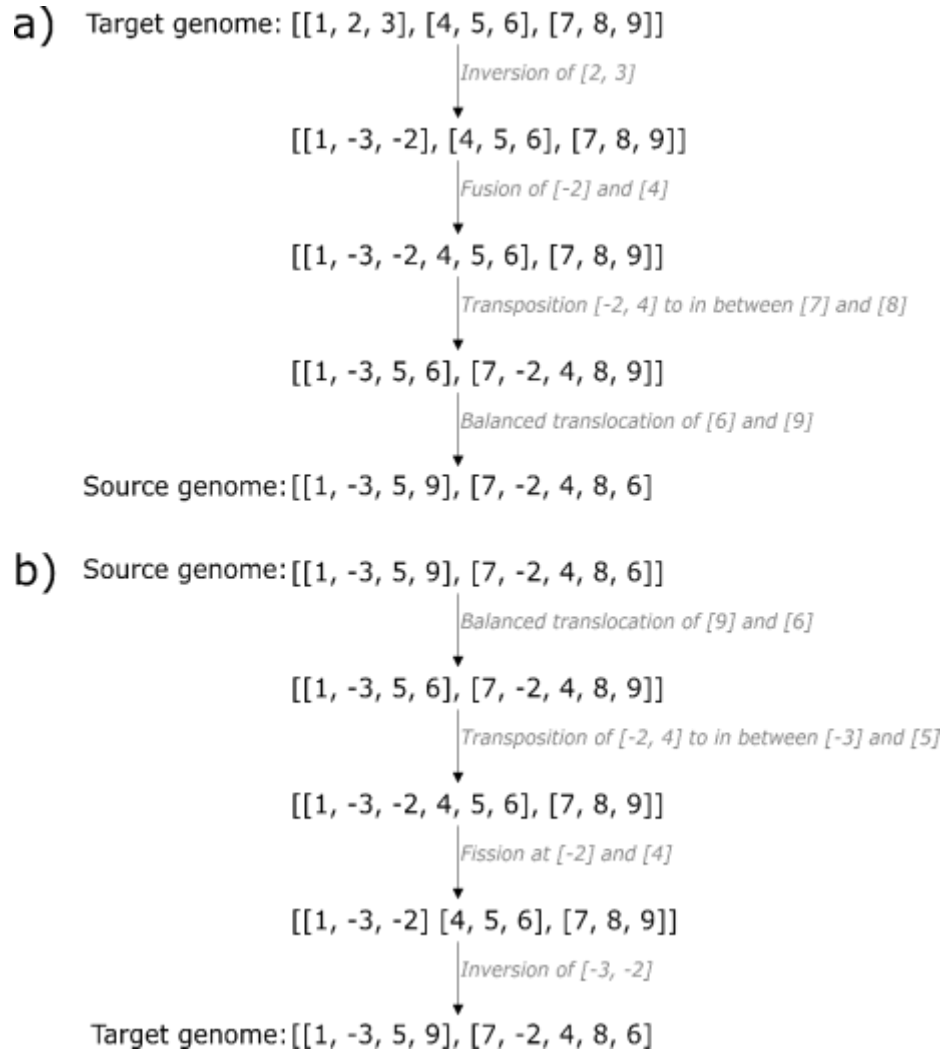


Fig 23. a) Illustration of how the Evolver generates a source genome from a target genome by applying a series of rearrangements to it. In this example, four rearrangement events are used for source genome generation. b) shows the inverse of the events in a) and describes the ‘true’ evolutionary scenario separating the source genome from the target genome.

Both the order of the original series of rearrangements and the locations (from which and to which sequence blocks move) are reversed (Fig 23b). For example, in Fig 23a, the third event is a transposition which comprises the movement of sequence blocks $[-2, 4]$ from in between $[-3]$ and $[5]$ to in between $[7]$ and $[8]$. In Fig 23b the inverse of this operation occurs in step 2 and consists of the movement of $[-2, 4]$ from in between $[7]$ and $[8]$ to in between $[-3]$ and $[5]$. In the case of inversions, transposition and balanced and unbalanced translocations, the *type* of rearrangement

remains the same when reversing the event. This is not the case for chromosome fusions and fissions, the inverse of a chromosome fusion is a chromosome fission and vice-versa (rearrangement event two in Fig 23a and rearrangement event three in Fig 23b).

When developing any tool or program, it becomes important to avoid the incorporation of biases. This is especially important when developing a tool to generate data that will be used to evaluate a different tool's performance. Biases within a data generation tool or program may result in certain skewed features in the data. When using this data to evaluate some other tool or simply to conduct a series of analyses, these features may, depending on their nature, be misattributed either as interesting phenomena or as biases within the tool that is being tested on the data.

For example, if the Evolver used one type of rearrangement more frequently than all the other during data generation, the analyses conducted on the output of Genolve using the data would most likely also show a disproportionate utilization of this one type of rearrangement. If we are unaware of the bias within the Evolver, the over-occurrence of the type of rearrangement would most likely be misattributed to built-in biases within Genolve.

To this end, the Evolver program, tasked with generating synthetic data, was developed in such a way as to maximize the number of variables that were assigned a pseudo-random value, albeit within a certain range.

The only input parameters for the Evolver is (i) the number of genes that the target genome should consist of and (ii) the number of rearrangements that is to be applied to the target genome in order to generate the source genome. The following variables were assigned pseudo-random values (using the 'random' built-in Python library)

1. The number of chromosomes
2. The number of genes per chromosome
3. The type of rearrangements applied
4. The locations of the rearrangements

It should be noted however that (i) the number of chromosomes is pseudo-randomly selected from within a range that scales with the number of genes, (ii) if there are four or fewer genes the genome will consist of only a single chromosome and (iii) although the type of rearrangements applied are pseudo-randomly selected, the selection of a rearrangement is made from the set of rearrangements

that are possible given the state of the genome at that point – those rearrangements that require two chromosomes in order to occur (balanced and unbalanced translocations and chromosome fusions) will not be available if the genome consists of a single chromosome at a given time point.

The output of the Evolver program thus includes a source-target genome pair as well as the ‘true’ evolutionary scenario that describes the evolution of the former into the latter

3.1.2 Genomic complexity notation

I refer to the specific number of rearrangements applied to the specific number of sequence blocks the genomes consist of as the *level or measure of genomic complexity* and denote it as

$$\text{number of rearrangements} \mid \text{number of sequence blocks}$$

This increases the probability that the source genome will be completely disordered relative to the target genome, *i.e.*, no final state adjacencies exist within the source genome generated. A higher number of sequence blocks present in the input genomes will have a negative impact on the runtime of Genolve. Thus, it is advantageous to collapse any sequence blocks that occur in consecutive order in the source genome, into a single sequence block.

See Fig 24 below. The adjacency between sequence blocks [-3, -2] in the source genome is (2.5, 3) (where the 2.5 is the head extremity of gene 2 and 3 is the tail extremity of gene 3 and the orientation of a gene is in the tail to head direction) which is a final state adjacency (*i.e.*, it is present in the target genome as well; See Fig 24, below). [-3, -2] in the source genome and [2, 3] in the target genome can thus be collapsed into a single sequence block (indicated in pink in the figure). Similarly, the adjacency between [4, 5] in the source genome, (4.5, 5), has an equivalent counterpart in the target genome. [4, 5] can thus also be collapsed into a single sequence block (shown in purple) resulting in input genomes that now consist of five instead of seven sequence blocks.

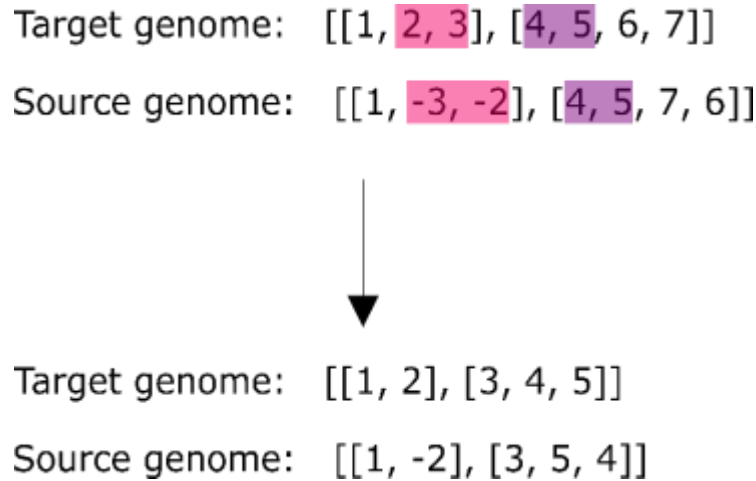


Fig 24. Illustration of how a source-target genome pair of which the source genome contains final-state adjacencies can be condensed. The adjacency (2.5, 3) in the target genome, between sequence blocks 2 and 3 (pink) is also present between sequence blocks -3 and -2 (pink). These two sequence blocks are condensed into a single sequence block in the bottom genomes. In purple at the top, the adjacency (4.5, 5) exists between sequence blocks 4 and 5 in both the target and source genome. These sequence blocks are thus also condensed into a single sequence block in the genomes at the bottom. This condensing of genomes results in a reduction in the number of sequences blocks making up the genome. At the top, the uncondensed genomes consist of seven sequence blocks each. In contrast, their condensed counterparts in the bottom of the figure consist of only five sequence block each.

3.2 Genolve's capability to solve source-target genome input pairs

It was important to establish whether Genolve is able to solve (successfully transform) any type of source-target genome pair that consists of the same set of sequence blocks. In total Genolve was tested on more than 4.8 million different source-target genome input pairs and was able to fully transform the source genome into the target genome 100 percent of the time. Table 1 shows the summary of these results.

Table 1. Shows the number of runs conducted at various level of genomic complexity and the percentage time Genolve was successfully transformed the source genome into the target genome.

Number of runs	Genomic Complexity	Percentage Time Genolve successfully transform the source genome into the target genome
800 000	3 4	100%
800 000	4 5	100%
800 000	5 6	100%
800 000	6 7	100%
800 000	7 8	100%
800 000	8 9	100%
800	1 20	100%
800	2 20	100%
800	3 20	100%
800	4 40	100%
800	5 20	100%
800	6 20	100%

3.3. Genolve's performance under different weighing ratios

During the development of Genolve, it was proposed that changing the weights of the different type of rearrangements to correspond with the frequency with which they are expected to occur would have a positive impact on Genolve's ability to include the 'true' series of rearrangements that resulted in the evolution of the source genome into the target genome in the solution set (See Section 2.3.7.). To investigate this, Genolve was tested on the same data under three different types of weighing ratios.

The first ratio, called the one-to-one ratio, is the default ratio Genolve utilizes. Under this ratio, all types of rearrangements have the same cost or edge weight in the network of solutions.

The second type of weighting ratio used, the same-as-solution ratio, was obtained using the Evolver's output. In addition to source and target genome pair, the Evolver also outputs the 'true' evolutionary scenario (the series of rearrangements that describes the source genomes synthetic evolution into the target genome). The frequency of occurrence of each of the different type of rearrangements within the 'true' evolutionary scenario was used to define the same-as-solution ratio. Those types of rearrangements that occur very rarely or not at all in the 'true' evolutionary scenario would have a higher cost or edge weight in the network of solutions. The same-as-solution ratio thus represents 'perfect knowledge' of the frequencies in which the different types of rearrangements are expected to occur for a set of input genomes.

Finally, Genolve was also analyzed under a pseudo-randomized ratio. This type of ratio was generated for each pair of input genomes by assigning each of the types of rearrangements a pseudo-random expected frequency of occurrence between 0 and 10. The addition of this ratio served as a control to ensure any interesting phenomena occurring under either of the former two weighting ratios do not occur under a simply pseudo-randomly assigned weighting ratio.

In addition to the three different types of weighing ratios, Genolve's ability to include the true evolutionary scenario in its output was evaluated across the following six measure of genomic complexity: 3|4, 4|5, 5|6, 6|7, 7|8, 8|9.

The graph in Fig 25 below shows the results of testing 100 000 different input genome pairs under each of the three types of weighing ratios at each of the six measures of genome complexities. The percentage time the 'true' evolutionary scenario was present in the results is plotted against the different measures of genomic complexity. Three differently coloured line plots are used to distinguish between the different types of weighting ratios.

At first glance, it is clear that the one-to-one weighting ratio outperforms the other two regarding the percentage time the 'true' evolutionary scenario is present in the solution set generated by Genolve. This was contrary to what was expected and may indicate that incorporating additional information regarding the expected frequency of occurrences of the different type of rearrangement events is not beneficial

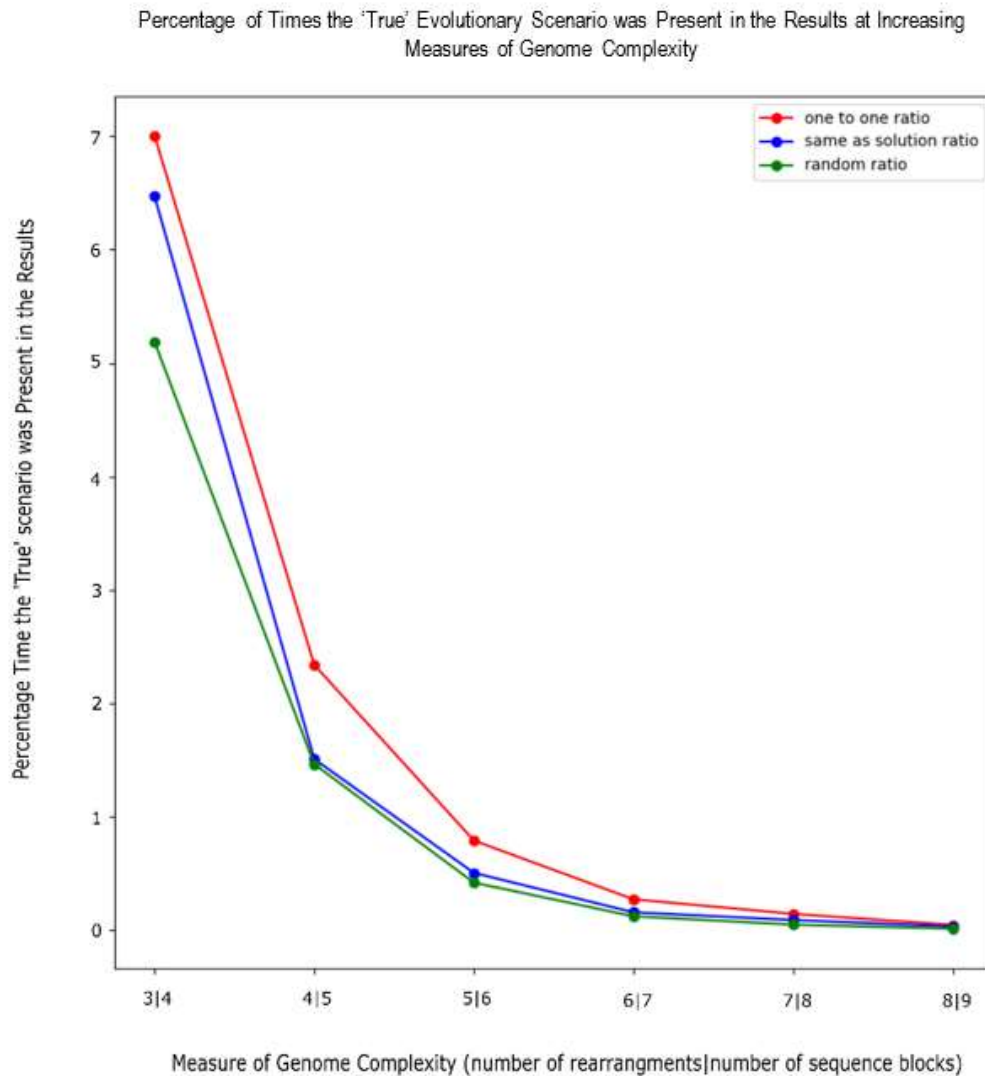


Fig 25. The percentage of times the 'true' evolutionary is present in the solution set generated by Genolve is plotted against increasing measures of genome complexity under a one-to-one weighting ratio (red), same-as-solution ratio (blue) and pseudo-randomized ratio (green). The percentage time the 'true' scenario was present in the solution set is given for 100 000 runs for each data point shown.

The second thing that stands out is how low the percentages values are. The maximum percentage of time the 'true' evolutionary scenario was present in the solution set generated by Genolve was 7% under the one-to-one ratio at a 3|4 genomic complexity level.

There are a number of explanations for these low values.

One explanation is the discrepancy in the number of rearrangements the Evolver uses to generate the source genome compared to the number of rearrangements Genolve uses to transform the source genome back into the target genome. As shown in Section 3.3.1, Genolve is often able to solve an input genome pair in fewer steps than was used by the Evolver to generate them. There are also a number of times Genolve uses more rearrangements than was used by the Evolver. In the pseudo-random environment created by the Evolver, it is possible that the same one or two sequence blocks are acted upon multiple times by rearrangement events resulting in a very convoluted evolutionary history. Unlike in a biological setting, where rearrangement events occur over large stretches of DNA, the Evolver acts across only a few integers representing sequence blocks. Numerous rearrangements affecting the same sequence blocks are expected to occur much more frequently in the simulated environment created using the Evolver than in a real biological setting. Despite this, DNA regions affected by multiple rearrangements *do* occur in genomic evolution, such regions are known as rearrangement hotspots. It is thus important to note that the accuracy of the rearrangement scenarios identified by Genolve for these regions is limited, and this should thus be taken into account when using the tool and analysing the results.

Another explanation for the low values of the percentage time the ‘true’ evolutionary scenario is present in the results can be the inability of Genolve to execute certain operations due to the underlying nature of the DCJ model. For example, Fig 26, similarly to Fig 23, shows a) how a source genome is generated from a target genome and b) what the ‘true’ evolutionary scenario, describing the transformation of source genome back into target genome, would be.

In Fig 26a, the final rearrangement executed by the Evolver is an inversion of sequence block [-7]. The first rearrangement that occurs in Fig 26b is the inversion of sequence block [7]. [7] is located between sequence blocks [3] and [-9]. The inversion of [7] will therefore destroy the adjacencies (3.5, 7) and (7.5, 9.5) and create the adjacencies (3.5, 7.5) and (7, 9.5).

The problem is that neither (3.5, 7.5) nor (7, 9.5) are final state adjacencies. If we refer back to how the DCJ model works (Section 2.3.2.), a DCJ operations will create *at least one* final state adjacency. Genolve will thus never identify the inversion of [7] as a valid operation, and as a result, the solution set will never contain the ‘true’ evolutionary scenario. This is a limitation of the DCJ model and by extension, a limitation of Genolve.

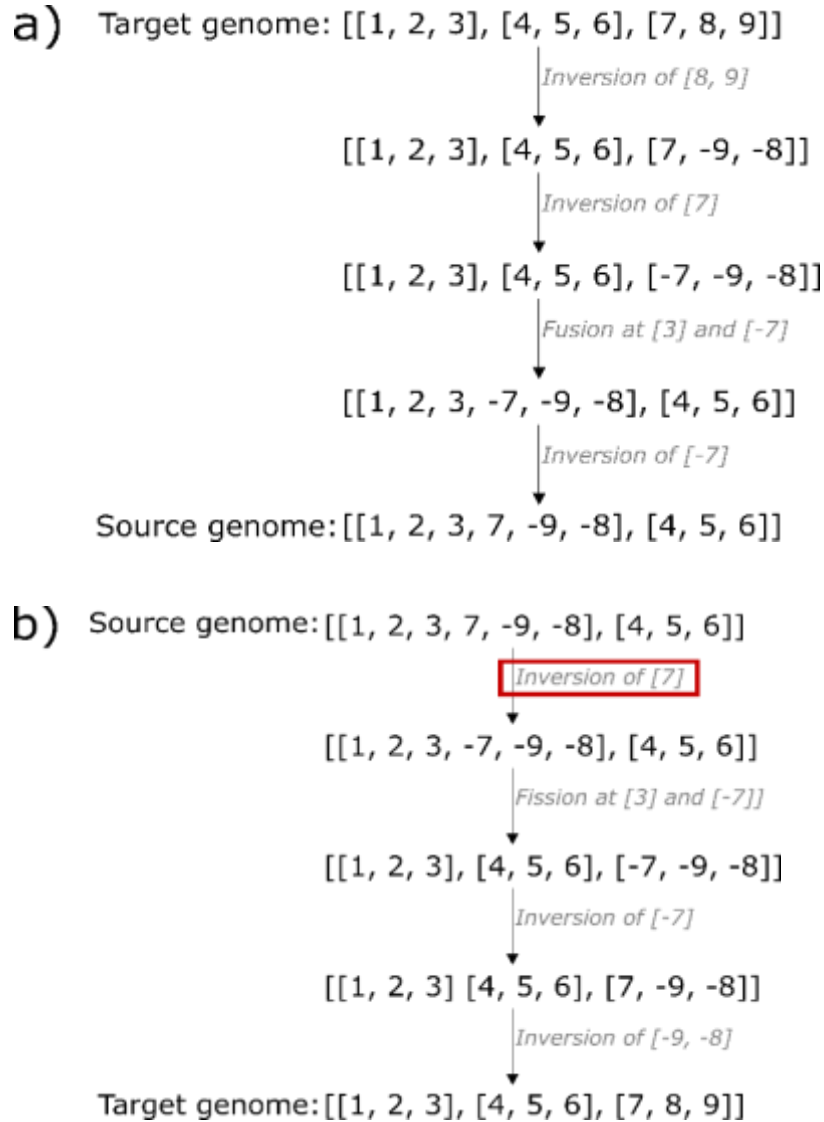


Fig 26. **a)** Illustration of how the Evolver generates a source genome from a target genome by applying a series of rearrangements to it. Four rearrangement events are used for source genome generation, the last of which is the inversion of sequence block $[-7]$. **b)** shows the inverse of the events in **a)** and describes the ‘true’ evolutionary scenario separating the source genome from the target genome. The first rearrangement is the inversion of sequence block $[7]$ (outlined in red). The inversion of $[7]$ creates the adjacencies $(3.5, 7.5)$ and $(7, 9.5)$. Neither of these are final-state adjacencies and the execution of the DCJ operation resulting in the inversion of $[-7]$ is therefore invalid.

Thirdly, it is possible that the Evolver applied two transpositions in such a way the Genolve had to solve it using a block interchange operation (see Section 2.3.5.). In this case, Genolve still identified the ‘correct’ rearrangements to resolve the particular area of sequence blocks if the

block-interchange operation was to be broken up into its individual components (i.e. the two transpositions).

The ability of Genolve to solve an input genome pair in fewer steps than was used in their generation can be attributable to the density of rearrangements used by the evolver. In a biological environment, a genome complexity of $3|4$ represents three rearrangement events applied to a whole chromosome worth of DNA sequence, so that the result of the rearrangements is four sequence regions that now occur in a different order relative to their ordering prior to the application of any rearrangements. When using the Evolver, we are not rearranging large DNA regions but rather a list of four integers. It is thus more likely that there will be a large amount of overlap (rearrangement events affecting the same sequence blocks/regions) when using the Evolver than there would be in a real biological environment, if we accept the principle of parsimony, *i.e.*, that the fewest number of steps to achieve a result is the best.

To evaluate to what extent overlapping rearrangements negatively influenced Genolve's ability to include the 'true' evolutionary scenario in its output, Genolve was run 100 times at the following levels of genome complexity: $1|20$, $2|20$, $3|20$, $4|20$, $5|20$, $6|20$.

It was expected that the percentage time the 'true' scenario was present in Genolve's output would (i) be much higher under a genomic complexity of $1|20$ than it was under a genomic complexity level of for example $3|4$, due to the decrease in rearrangement density and (ii) decrease as the level of genomic complexity increased. The results in Fig 27 show that this was the case.

Decreasing the rearrangement density made a clear difference: the number of times the 'true' solution was present in the solution set generated by Genolve was strikingly higher. As expected, it did decrease as the level of genomic complexity (and by extension the density of rearrangement operations), increased.

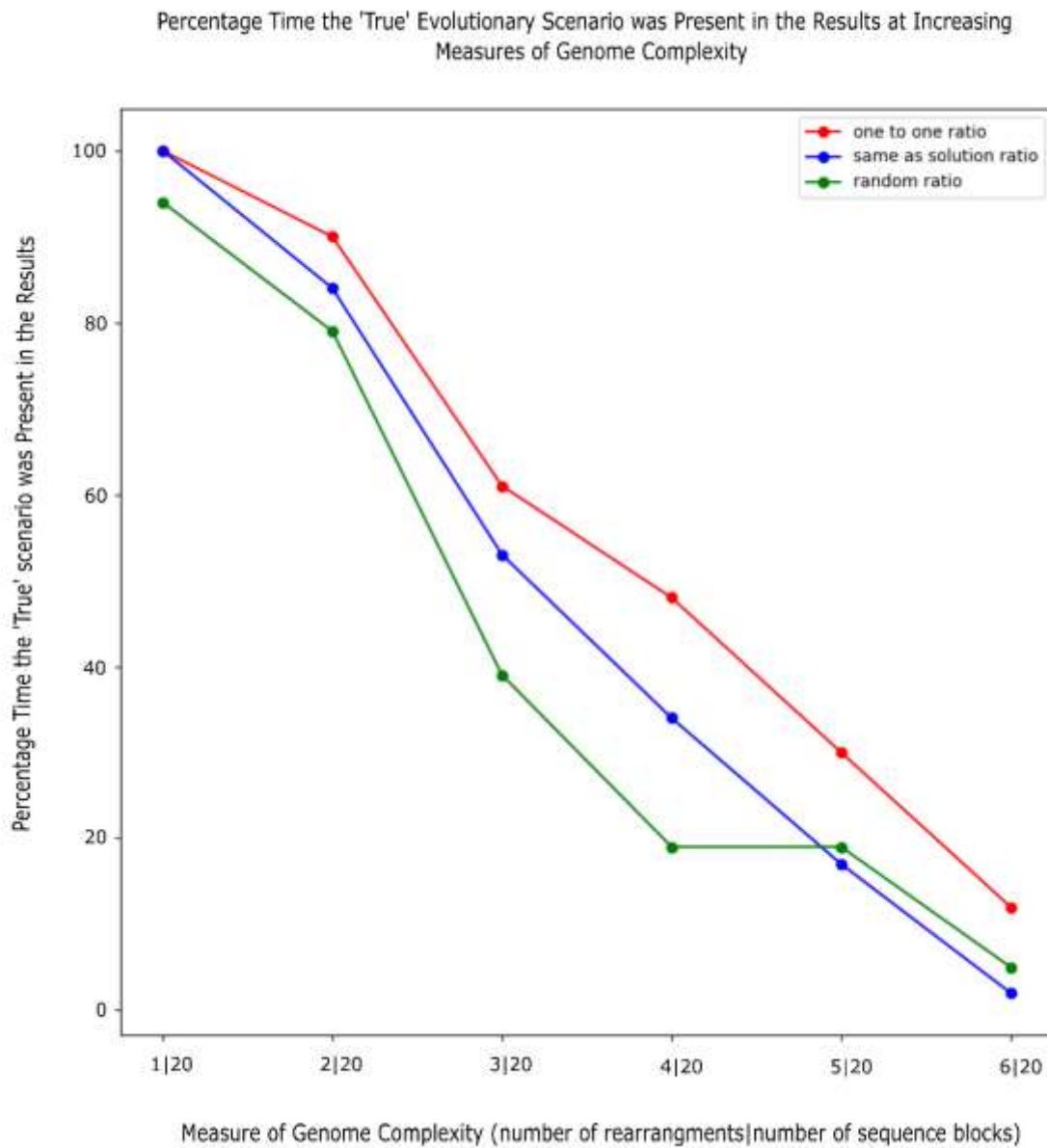


Fig 27. The percentage time the 'true' evolutionary is present in the solution set generated by Genolve is plotted against increasing measure of genome complexity under a one-to-one weighting ratio (red), same-as-solution ratio (blue) and pseudo-randomized ratio (green). The percentage time the 'true' scenario was present in the solution set is given for 100 runs for each data point shown.

A high operation density can explain all three of the factors previously given to explain why the percentage time the 'true' evolutionary scenario is present in the results may be so low, namely:

1. Genolve does not use the same number of rearrangements to transform source genome into the target genome as was used by the Evolver to generate source genome from the target genome.
2. The ‘true’ evolutionary scenario given by the Evolver contains rearrangements for which no legal DCJ operation exists (refer to the example in Fig 26).
3. The Evolver makes use of two transpositions in such a way the Genolve can only solve it using a block-interchange operation (refer to the example in Fig 19).

All of the above three factors will prevent the inclusion of the ‘true’ evolutionary scenario in Genolve’s output.

This is a critical factor to keep in mind. Despite the general low frequency of occurrence of genome rearrangements during genome evolution, there has been evidence supporting the existence of rearrangement hotspots in certain species [68]. If hotspots do exist in input genomes analysed with Genolve, Genolve will likely be unable to give an accurate depiction of the real evolutionary events that occurred due to the three factors mentioned above.

3.3 The size of solution sets and the length of solutions

As genomic complexity increases, both the number of solutions within a solution set and the average length of the solution (*i.e.*, number of rearrangements it consists of) are expected to increase. The more rearrangements used by the Evolver to generate the source genome, the more rearrangements will be necessary to transform the source genome back into the target genome.

Similarly, the higher the number of independent rearrangements (rearrangement that do not overlap) separating two genomes, the larger the number of permutations of rearrangement events that can describe the transformation of the source genome into the target genome.

3.3.1 The average number of solutions per solution set

It was important to ascertain the quantity of solutions within a solution set that can be expected as increases in solution set size also increases the runtime of Genolve. If a solution set is too large, it may become practically incomputable.

The sizes of solution sets were evaluated at increasing levels of genomic complexity: 3|4; 4|5; 5|6; 6|7; 7|8, 8|9. Fig 28 below shows a semi-log plot of the average number of solutions within a solution set over 100 000 runs *versus* the measure of genome complexity. The red plot shows the results under a one-to-one weighing ratio, the blue plot under a same-as-solution weighing ratio, and the green under a pseudo-random weighing ratio.

The general trend under all three weighing ratios is exponential, indicating a non-linear relationship between the number of solutions expected in a solution set and increases in genomic complexity. There is a difference in the number of solutions between the three weighing ratios. These differences are attributable to two factors:

1. Under a one-to-one ratio, all rearrangements have the same cost and those solutions included in a solution set will simply be all the shortest paths (those containing the fewest number of operations). Under the same-as-solution and pseudo-randomized ratios, the costs of rearrangements differ. This may result in the exclusion of some solutions which contain very high cost rearrangements.
2. Alternatively, the cost difference between rearrangements under the same-as-solution and pseudo-randomized weighing ratios may lead to the inclusion of previously excluded solutions where, despite containing many operations, the operations a solution consists of, all have very low costs, which results in the solution not exceeding the minimum cost cut off for its particular solution set.

In the graph we can see that of the two factors mentioned above, the first was predominant, with the solution set sizes under the one-to-one weighing ratio being higher at all expect one level of genomic complexity. At the final level of genomic complexity, 8|9, the average number of solutions per solution set is slightly higher under the same-as-solutions ratio than under the one-to-one ratio, suggesting the second factor played a larger role.

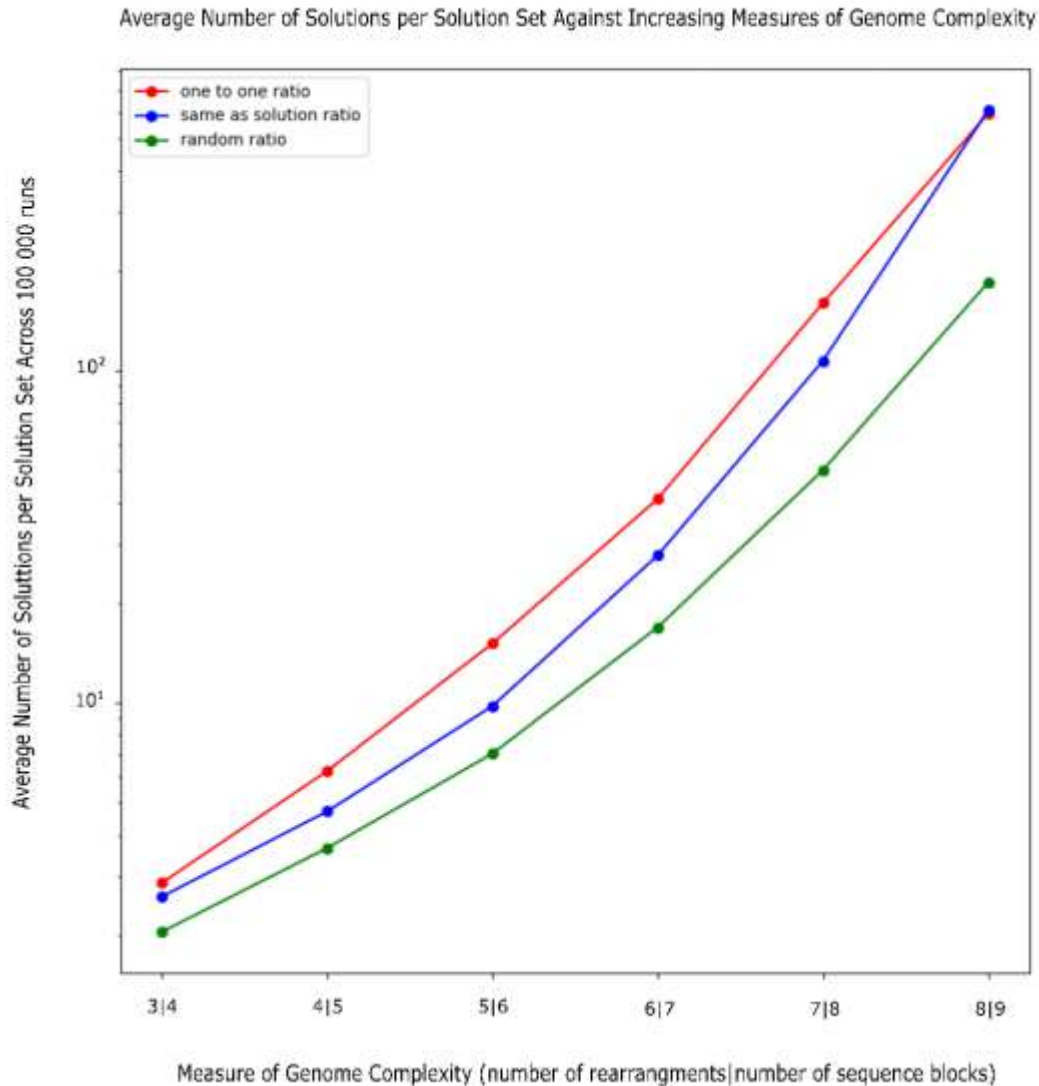
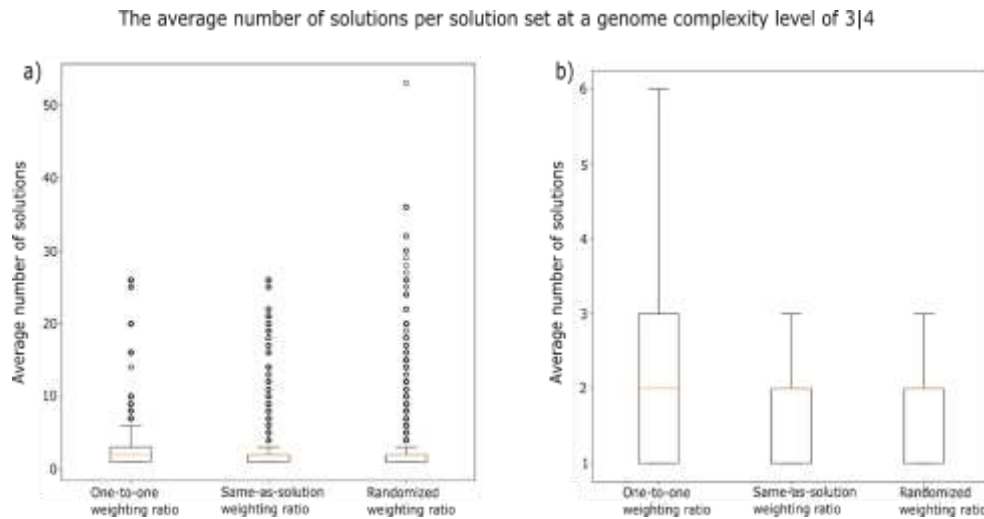


Fig 28. The number of solutions within a solution set is plotted at increasing levels of genomic complexity. Each data points shows the average number of solutions within a solution set across 100 000 runs. The results under each of the different weighting ratios, one-to-one ratio (red), same-as solution ratio (blue) and pseudo-randomized ratio (green) are plotted on the graph.

The data points in the figure show the average value across 100 000 runs. Within these 100 000 runs, there was often a fair amount variation in the number of solutions per solutions set. To illustrate the spread of the data within the 100 000 runs box-and-whisker plots for each of the three weighting ratios are given at the six different levels of genomic complexity (Fig 29.1 – Fig 29.6).

The often-vast number of outliers, defined as exceeding $Q3 + 1.5 \times IQR$ or falling below $Q1 - 1.5 \times IQR$, where $Q1$ is the first quartile, $Q3$ is the third quartile, and IQR is the interquartile range ($Q3 - Q1$) makes it difficult to see the box-and-whisker component of the plots. To resolve this, for each of the figures, the entire plot, including outliers is shown in a) and the enlarged box-and-whisker component, where the outliers have been excluded, is shown in b).

Each graph shows the results under each of the three weighting ratios (namely: one-to-one ratio, same-as-solution ratio and randomized ratio) under the six different genomic complexity levels.



*Fig 29.1. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 3/4 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3 - Q1$). The orange line indicates the median of each data set.*

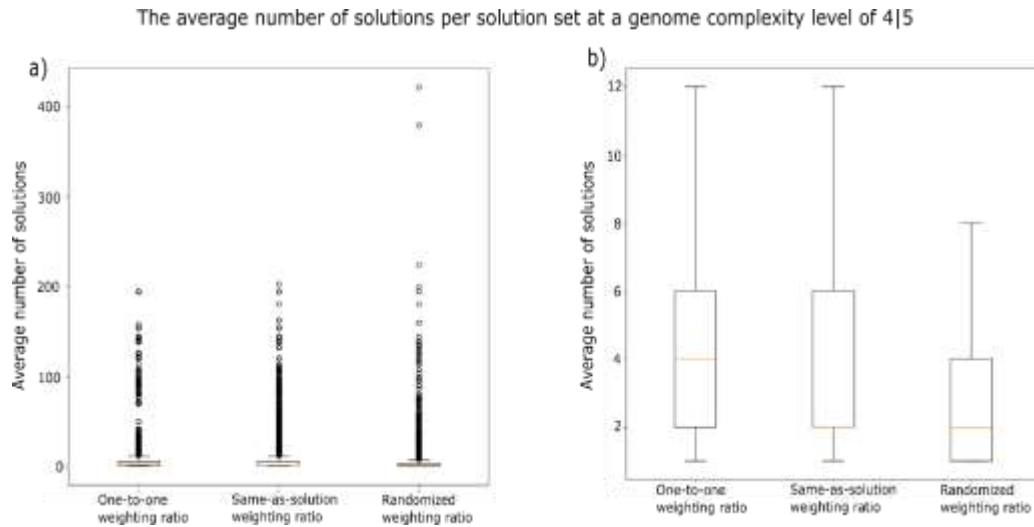


Fig 29.2. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 4|5 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by 1.5 x the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

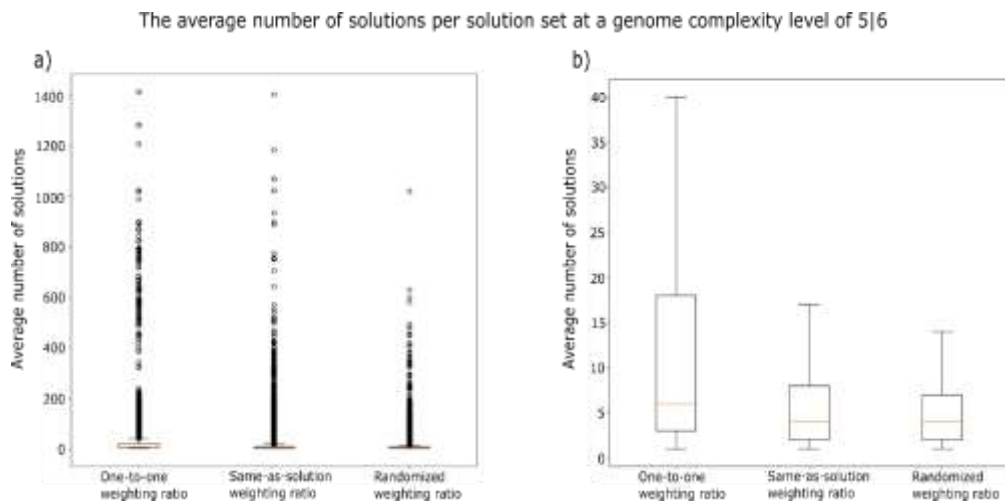


Fig 29.3. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 5|6 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by 1.5 x the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

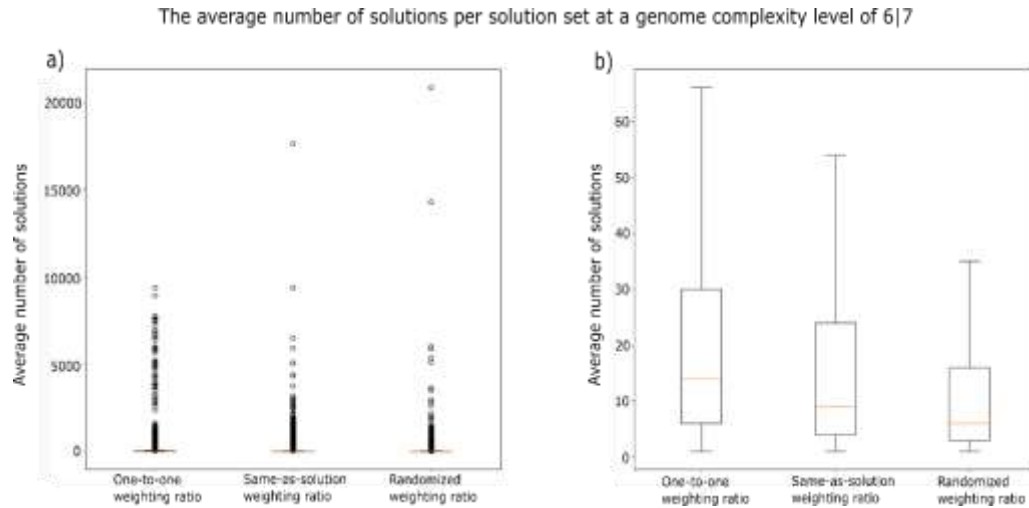


Fig 29.4. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 6|7 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by 1.5 x the interquartile range ($Q3-Q1$). The median of each data set is indicated by the orange line.

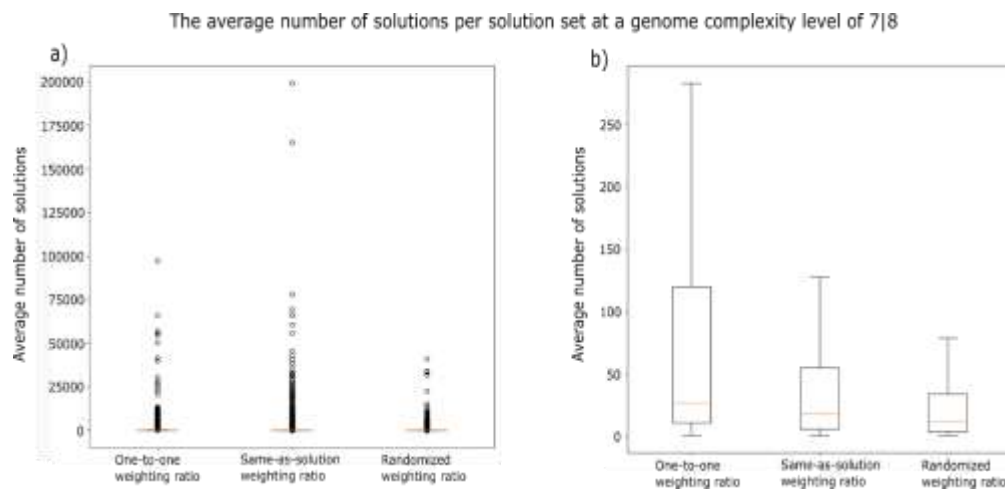


Fig 29.5. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 7|8 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by 1.5 x the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set..

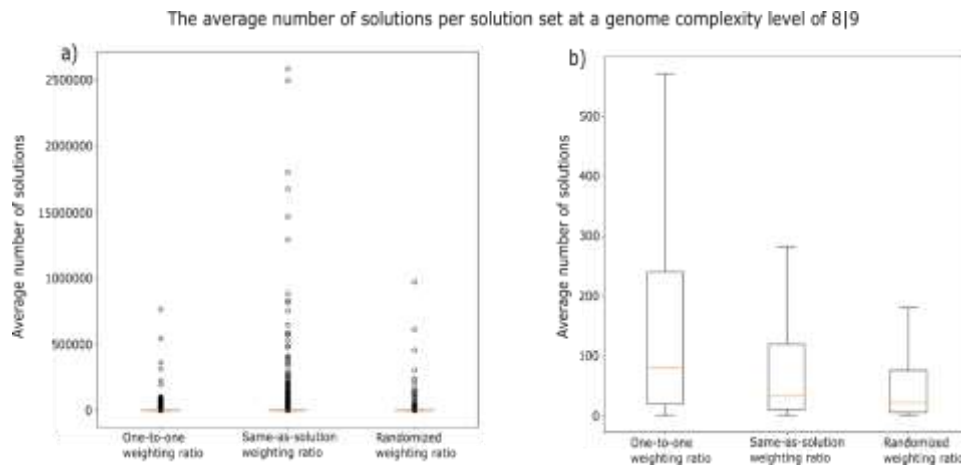


Fig 29.6. Box-and-whisker plot of the number of solutions per solution set under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 8|9 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

The large variation within each of the 100 000 runs was not unexpected, as some evolutionary scenarios will be more complex than others, depending on the order, type and locations of rearrangement events.

A large number of independent rearrangements (rearrangements that do not affect the same sequence blocks) will result in a larger solution set as a solution for each type of permutation will exist, *i.e.*, the solution set will contain a solution for each of the different permutations of independent rearrangements. An example is given in Fig 30, which shows the solution set generated by Genolve for the following input genomes:

Source genome: `[[1, -2, 3, 4, 6, 5]]`

Target genome: `[[1, 2, 3], [4, 5, 6]]`

Three independent rearrangements are separating the two input genomes namely an inversion of sequence block [2], a chromosome fusion/fission at sequence blocks [3] and [4] and lastly a transposition of sequence block [6] to in between sequence blocks [4] and [5].

```
#####

Source Genome: [[1, -2, 3, 4, 6, 5]]
Target Genome: [[1, 2, 3], [4, 5, 6]]

Number of most parsimonious solutions: 6

Solutions:

Solution number 1 Tansposition - Fission - Inversion
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, -2, 3, 4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, -2, 3, 4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))
([[1, -2, 3], [4, 5, 6]], ('fis', ((3.5, 4), 3.5, 4)))
([[1, 2, 3], [4, 5, 6]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))

Solution number 2 Fission - Transposition - Inversion
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, -2, 3], [4, 6, 5]], ('fis', ((3.5, 4), 3.5, 4)))
([[1, -2, 3], [4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, -2, 3], [4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))
([[1, 2, 3], [4, 5, 6]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))

Solution number 3 Tansposition - Inversion - Fission
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, -2, 3, 4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, -2, 3, 4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))
([[1, 2, 3, 4, 5, 6]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))
([[1, 2, 3], [4, 5, 6]], ('fis', ((3.5, 4), 3.5, 4)))

Solution number 4 Inversion - Tansposition - Fission
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, 2, 3, 4, 6, 5]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))
([[1, 2, 3, 4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, 2, 3, 4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))
([[1, 2, 3], [4, 5, 6]], ('fis', ((3.5, 4), 3.5, 4)))

Solution number 5 Fission - Inversion - Transposition
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, -2, 3], [4, 6, 5]], ('fis', ((3.5, 4), 3.5, 4)))
([[1, 2, 3], [4, 6, 5]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))
([[1, 2, 3], [4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, 2, 3], [4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))

Solution number 6 Inversion - Fission - Transposition
([[1, -2, 3, 4, 6, 5]], ('none, this is the source genome', 'N/A'))
([[1, 2, 3, 4, 6, 5]], ('inv', (((1.5, 2.5), (2, 3)), ((1.5, 2), (2.5, 3)))))
([[1, 2, 3], [4, 6, 5]], ('fis', ((3.5, 4), 3.5, 4)))
([[1, 2, 3], [4, 5], ['o', 6]], ('trp0', (((4.5, 6), (5, 6.5)), ((4.5, 5), (6, 6.5)))))
([[1, 2, 3], [4, 5, 6]], ('trp1', (((6, 6.5), 5.5), ((5.5, 6), 6.5))))

#####
```

Fig 30. Output generated by the Genolve tool for the following input genome pair, source genome: $[[1, 2, 3, 4, 6, 5]]$, target genome: $[[1, 2, 3], [4, 5, 6]]$. The solution set contains six solutions, consisting of the same three independent rearrangements, an inversion, a chromosome fission and a transposition. The six solutions are the different permutations, $3!$, of these rearrangements. The ordering of the rearrangement for each solution is shown in red next to the solution number.

The solution set generated by Genolve, includes six solutions, each of which contain the above mentioned three rearrangements in the six different orders they can occur.

To see how solution set size scale with the number of independent solutions, Genolve was run on genomes separated by an increasing number of independent rearrangements.

Table 2 below shows the number of solutions generated by Genolve for an input genome pair separated by n number of independent rearrangements (and no dependent rearrangements).

Table 2. The number of solutions present within the solution set generated by Genolve for input genome pairs separated by an increasing number of independent rearrangements.

Number of independent rearrangements (n)	Number of solutions generated by Genolve
1	1
2	2
3	6
4	24
5	120
6	720
7	5041
8	40320

Upon closer examination of how the number of solutions increases with increases in the number of independent rearrangements, the factorial relationships between the number of independent rearrangements and the number of solutions within the solution set generated by Genolve become apparent:

For an input genome pair separated by, and only by, n independent rearrangement events, there will be $n!$ solutions within a solution set.

Another factor that would increase the size of a solution set is combinations of different types of rearrangement events that achieve the same outcome.

For example, Fig 31 shows the solution set generated by Genolve for the following input genomes:
Source genome: [[1, -3, -2, 4], Target genome: [[1, 2], [3, 4]]


```
#####
Source Genome: [[1, -3, -2, 4]]
Target Genome: [[1, 2], [3, 4]]

Number of most parsimonious solutions: 2

Solutions:

Solution number 1
([[-3, -2, 4]], ('none, this is the source genome', 'N/A'))
([[-3], [-2, 4]], ('fis', ((2.5, 3), 2.5, 3)))
([[-2], [3, 4]], ('b_trl', (((1.5, 3.5), (2, 4)), ((1.5, 2), (3.5, 4)))))

Solution number 2
([[-3, -2, 4]], ('none, this is the source genome', 'N/A'))
([[-3, 2, 4]], ('inv', (((1.5, 3.5), (2, 4)), ((1.5, 2), (3.5, 4)))))
([[-2], [3, 4]], ('fis', ((2.5, 3), 2.5, 3)))

#####
*Where inv = inversion, b_trl = balanced translocation, fis=fission
```

Fig 31. Output generated by the Genolve tool for the following input genome pair, source genome: $[[1, -3, -2, 4]]$, target genome: $[[1, 2], [3, 4]]$. The solution set contains two solutions, the first of which consists of a chromosome fission followed by a balanced translocation, and the second of an inversion followed by a chromosome fission. The symbols of the different rearrangement events are underlined in red.

The same outcome is achievable both by applying a fission followed by a balanced translocation or by an inversion followed by a fission, and therefore both scenarios will be included in the solution set of Genolve. The more rearrangements there are affecting the same sequence blocks, the higher the likelihood that there are different combinations of

Referring back to Fig 29.1 through Fig 29.6, the upper whisker of the plots for the one-to-one weighting ratios is generally larger than that of the same-as-solution and pseudo-randomized weighting ratios. This corresponds with the average number of solutions per solution set being higher under the one-to-one ratio than for the other two in Fig 28. As previously discussed, an exception occurs at an 8|9 genomic complexity level, where the average number of solutions per solution set is largest under the same-as-solution weighting ratio. The spread of the data points at this complexity level is shown in Fig 29.6. At the shown level of genomic complexity, there are a great many more outliers that pull the mean higher.

3.3.2. *The average length of a solution*

The average length of a solution within a solution set was also investigated. One hundred thousand runs were conducted at the following levels of genomic complexity: 3|4, 4|5, 5|6, 6|7, 7|8, 8|9, under each of the three weighing ratios.

Fig 32.1 to Fig 32.6 show scatter plots of the average length of the solutions within a solution set per run. In each figure, the red scatter plot shows the results under the one-to-one ratio, the blue under the same-as-solution ratio, and the pseudo-randomized ratio are shown in green.

Examination of these scatter plots immediately spark the following questions:

- (i) Why are there clear horizontal lines within each of the graphs?
- (ii) Why do the same-as-solution and pseudo-randomized ratio plots contain additional scattered points between the horizontal lines and the one-to-one ratio plot does not?
- (iii) Why are some of the average solution lengths higher than the number of rearrangements present in the 'true' evolutionary scenario?
- (iv) Why are many of the average solution lengths lower than the number of rearrangements present in the 'true' evolutionary scenario?
- (v) Why is the maximum average solution length under the pseudo-randomized ratio consistently higher than under the same-as-solution ratio and the latter consistently higher than under the one-to-one ratio?

Firstly, horizontal lines are visible on the plots because the number of runs is so high. The results for 100 000 runs are shown on a single plot, and individual data points are thus located so closely together as to appear as a horizontal line if there is a sufficient number of solution sets with a specific average solution length. The clearer or more solid a line appears; the more solutions sets with that specific average solution length was present in the data set.

The one-to-one ratio plots have data points located solely at whole numbers, which is not the case under the other two ratios. This is to be expected due to the nature of the weighting ratios. All solutions within a solution set under a one-to-one ratio have the same length (since all rearrangements have the same cost and only the length of a path determines whether it is included in the solution set), thus the average of all solution lengths is equivalent to any one solution length. This is not the case under the other two weighting ratios, where solutions within the same solution

set can have varying lengths. The average solution length of solutions within a solution set may thus often be a fraction.

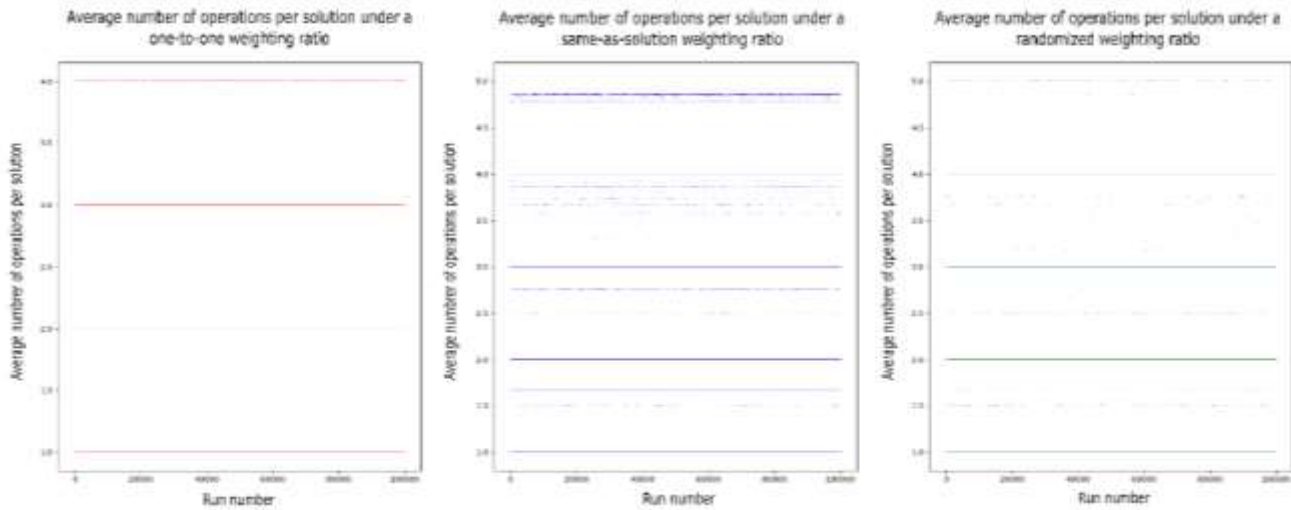


Fig 32.1. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of $3/4$. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

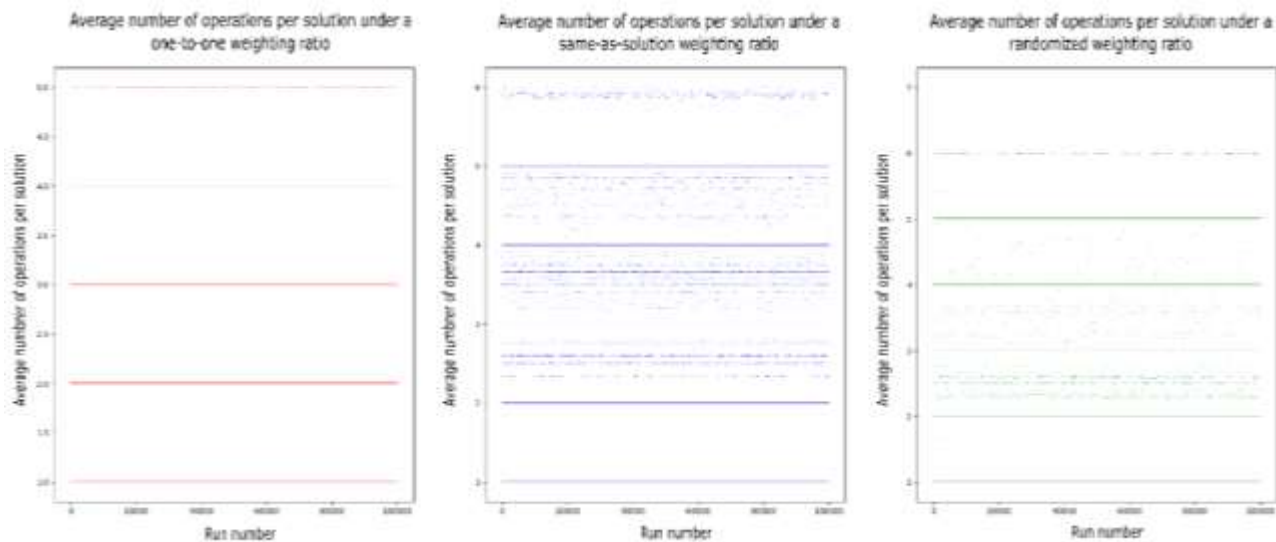


Fig 32.2. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of $4/5$. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

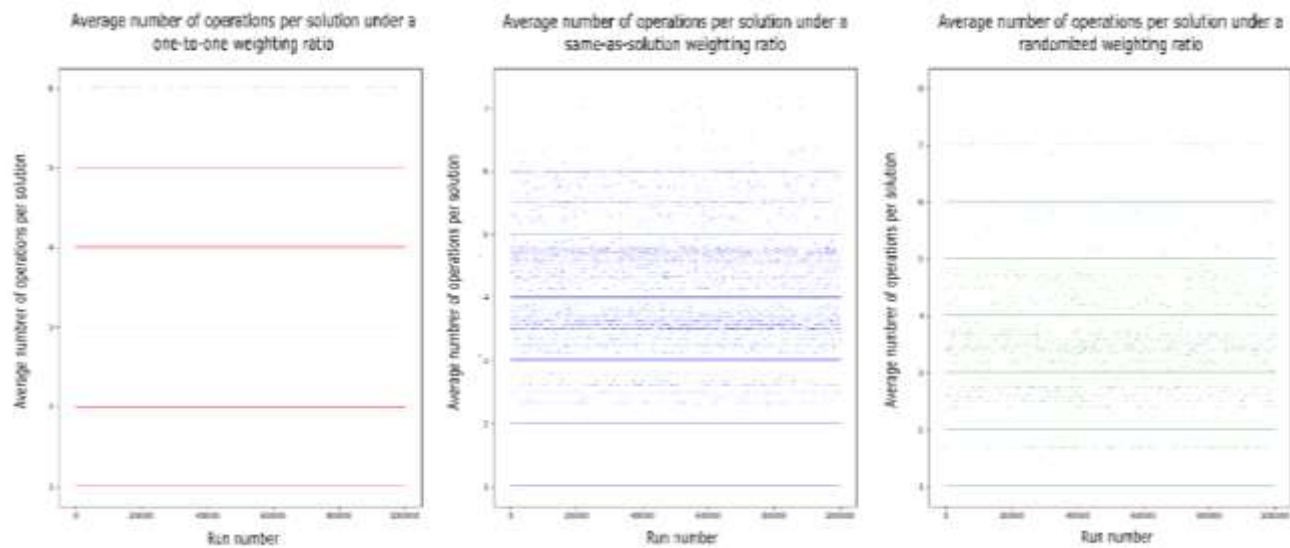


Fig 32.3. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 5/6. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

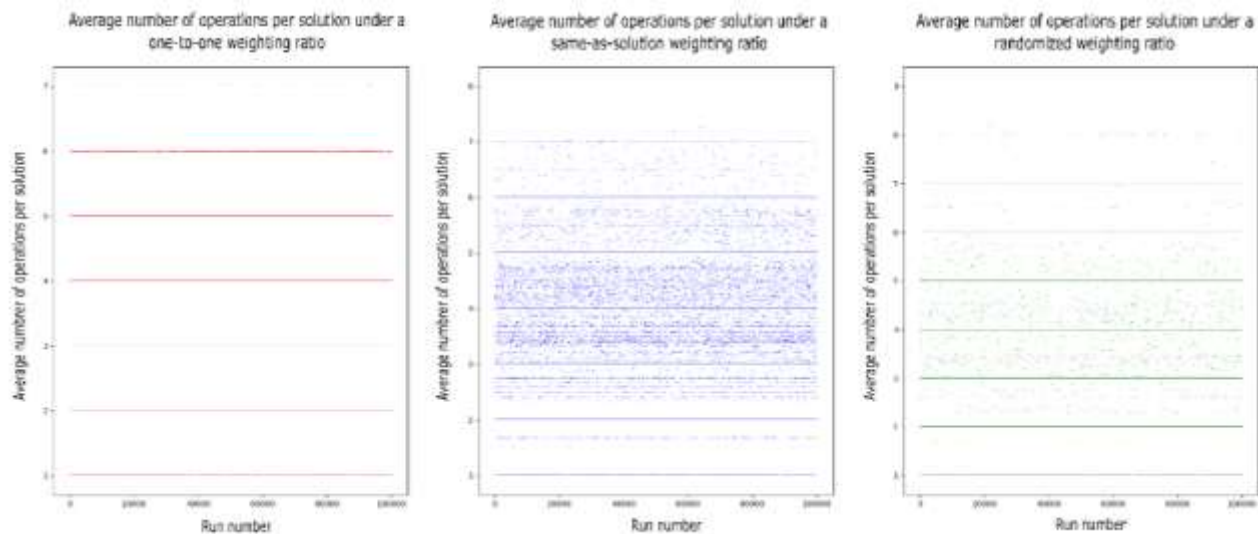


Fig 32.4. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 6/7. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

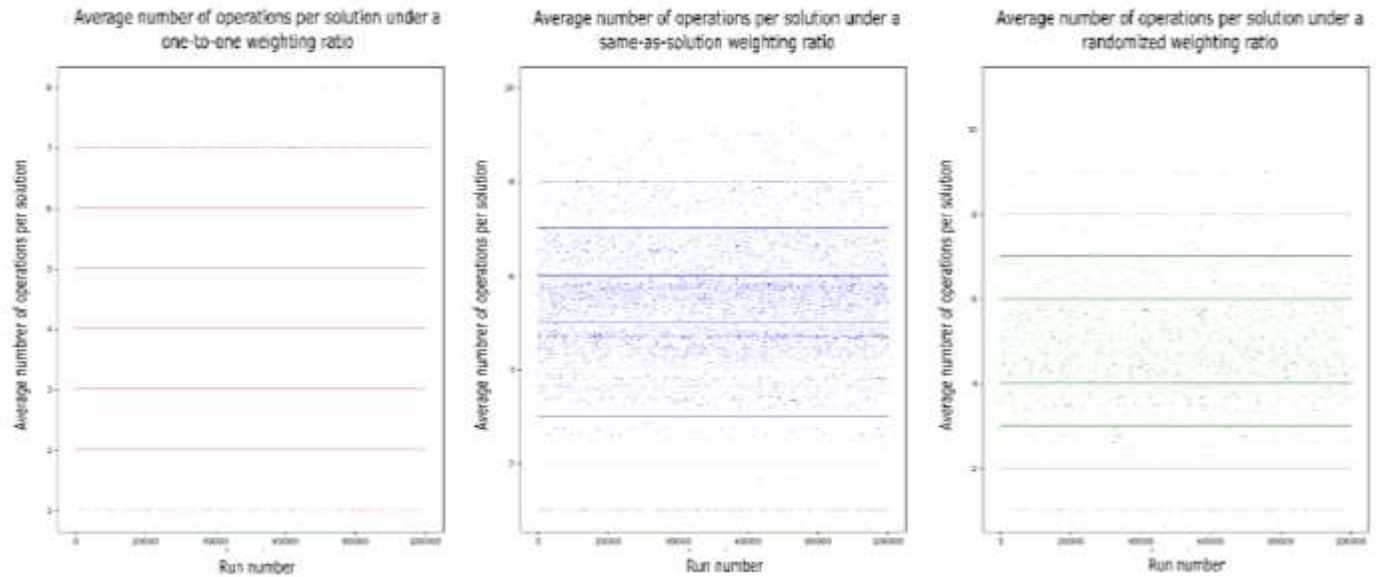


Fig 32.5. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 7/8. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

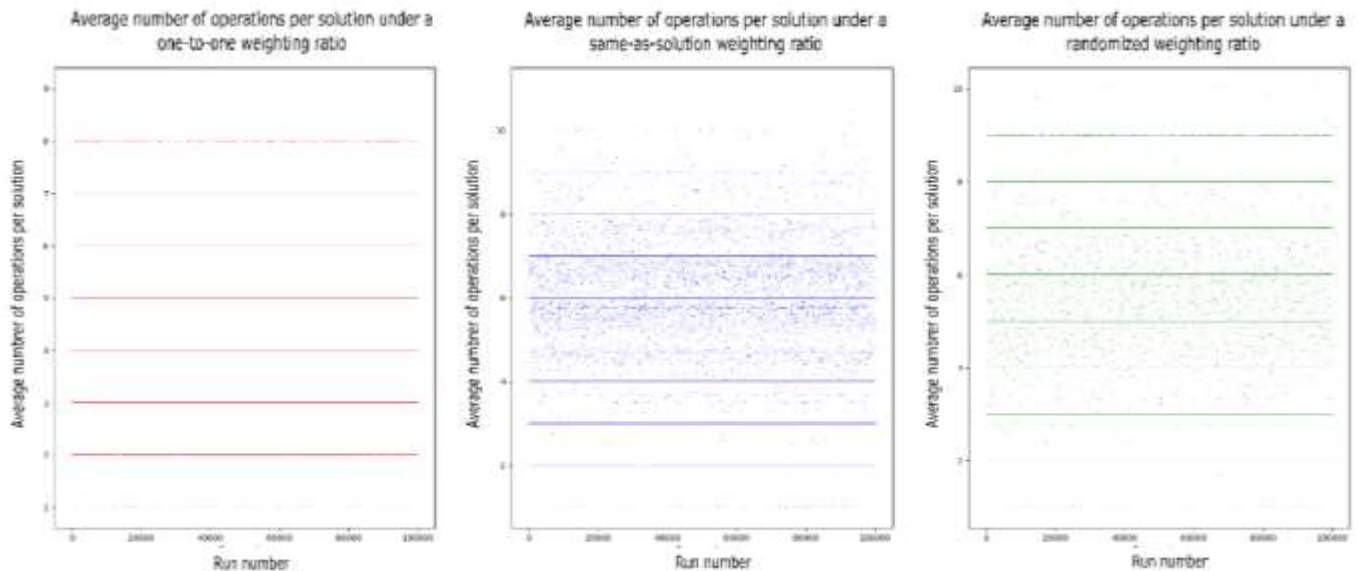


Fig 32.6. Scatter plots of the average length of a solution within a solution set for each of the individual 100 000 runs at a genomic complexity level of 8/9. The red, blue and green plots give the results under a one-to-one, same-as-solution and pseudo-randomized weighting ratio, respectively. Solid lines indicate a large number of runs for which the average solution length was equal to a particular value.

At all levels of genomic complexity, under each of the different weighting strategies, there were solution sets with an average solution length higher than that of the ‘true’ evolutionary scenario and numerous solution sets with an average solution set lower than that of the ‘true’ evolutionary scenario.

As mentioned in Section 3.3, the pseudo-random environment created by the Evolver often results in the same one or two sequence blocks being affected by numerous rearrangement events resulting in a convoluted evolutionary history. Although this is expected to occur less frequently in a biological environment where rearrangements occur over entire genomes instead of a few integers representing sequence blocks, it *is* known to occur in nature [68]. Although Genolve is still able to identify possible rearrangement scenarios that can account for the evolutionary history in these rearrangement hotspot regions, the tool’s accuracy is expected to be lower than for regions not affected by numerous rearrangements.

Due to the pseudo-random nature by which the Evolver executes rearrangements, there would be cases where a rearrangement it applies reverses some rearrangement applied previously. An example is shown in Fig 33. The Evolver applies two inversions and a transposition to the target genome, but the two inversions affect the same sequence block. Therefore, the transformation of the source genome back into the target genome requires but a single transposition, whilst the ‘true’ evolutionary scenario will contain three rearrangements.

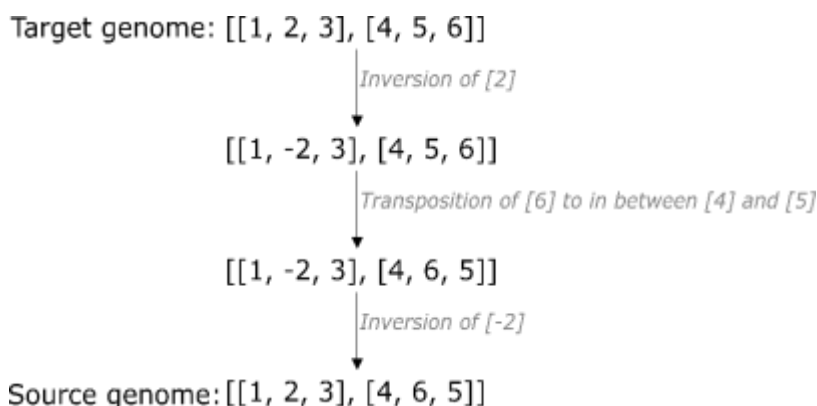
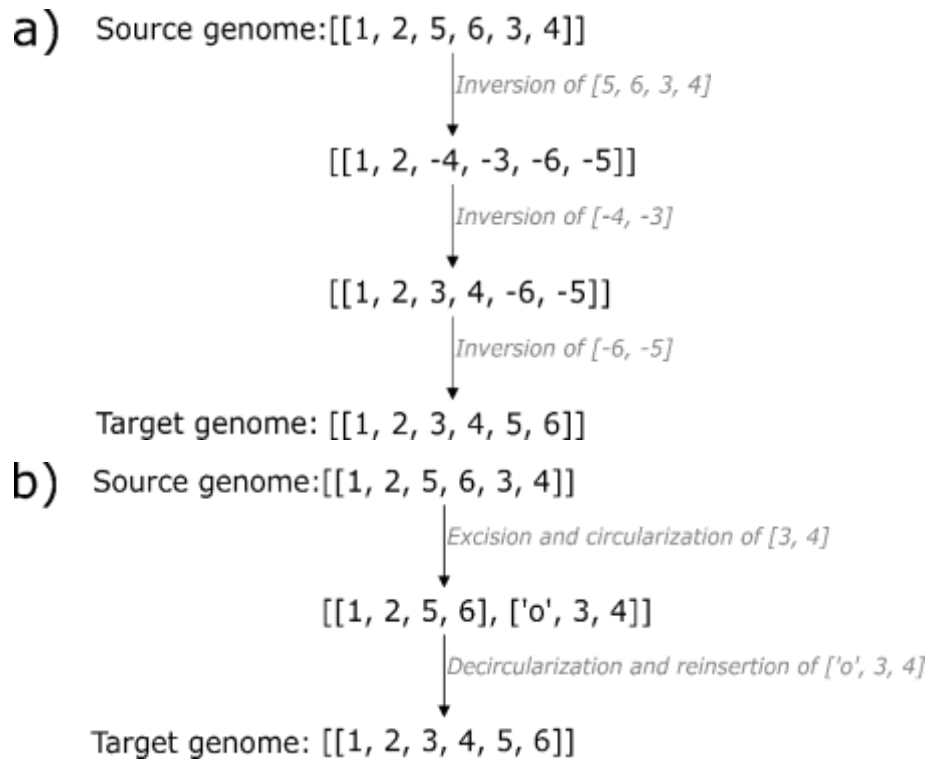


Fig 33. The generation of a source genome from a target genome. Three rearrangements are applied to the target genome. The effects of the first rearrangement, an inversion of sequence block [2] is reversed by the application of the final rearrangement, an inversion of sequence block [-2].

It is also possible that a number of rearrangements applied to the same sequence blocks of a target genome by the Evolver could result in a situation where Genolve is able to solve it using fewer and often different rearrangements.

Fig 34 shows an example of a case where Genolve transformed the source genome into the target genome in fewer steps than was used to generate the source genome. The ‘true’ evolutionary scenario generated by the Evolver is shown in Fig 34a where three inversions are used for transformation of the source genome into the target genome. Genolve was, however, able to solve the same pair of input genome using a single transposition (Fig 34b).



*Fig 34. An illustration of how the same source-target genome pair can be solved using alternatively **a)** three successive inversion events or **b)** a single transposition event.*

The final question raised regarding the results in Fig 32.1 to 32.6, was that the maximum average solution length under a one-to-one weighting ratio was consistently lower than that of the other two ratios at each of the levels of genomic complexity. A similar case is observed when comparing

the results under a same-as-solution weighting ratio to that of a pseudo-randomized weighting ratio. This phenomenon is linked to the different weights placed on rearrangement types under the different weighting ratios. If one rearrangement type has a significantly lower cost than another, Genolve may use three of the cheaper rearrangements instead of one of a more expensive rearrangements, increasing the total number of rearrangements used. For example, in Fig 34, both scenarios would be identified by Genolve, but under a one-to-one weighting ratio, only the scenario in Fig 34b would appear in the solution set generated by the tool.

Under a different weighting ratio where a transposition has a cost of three times that of an inversion, both scenarios in Fig 34 would be present in the output of Genolve. Of course, if transpositions cost more than three times that of an inversion, only the scenario in Fig 34b would be present in the solution set.

3.6 The runtime and memory usage of Genolve

The runtime and memory usage of Genolve were the final two factors investigated. These are important factors to consider because they indicate the tool's capabilities, how long it may be necessary to wait for results, and what resources are required to utilize the tool. All analyses for this thesis were run on the high-performance computer cluster (HPC2) of Stellenbosch University.

3.6.1 The runtime of Genolve

Analyses of the time taken by Genolve to generate a solution set for a single pair of input genomes were done at the following levels of genomic complexity: 3|4, 4|5, 5|6, 6|7, 7|8, and 8|9. At each of these complexities, 100 000 runs were executed, and the average across the runs plotted on the graph in Fig 35, below. The red, blue and green plots correspond to the results under a one-to-one, same-as-solution and pseudo-randomized ratio, respectively.

The increase in runtime is relatively severe and is exponential as the level of genome complexity increases. If Genolve is run on the *same* input genome pairs under each of the different weighting ratios, it is expected, in theory, that the time Genolve requires, *up until conclusion of network construction*, will be identical for each of the different weighting ratios. The variations between the results for the weighting ratios should thus only be attributable to the time taken to *traverse*

the network of solutions and generate the set of lowest cost solutions. In reality, another factor will influence the variance between the different weighing systems, namely the variability in processing speed of the CPU that is running the script.

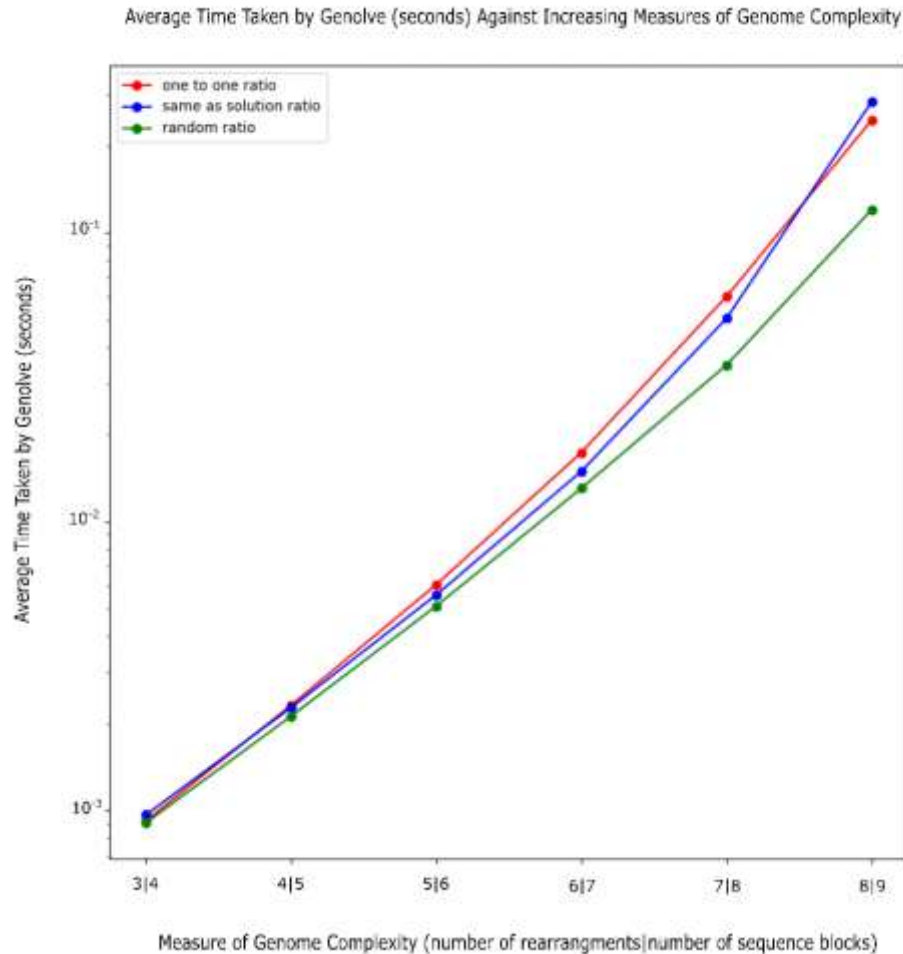


Fig 35. The average time required by Genolve to generate a solution set for a pair of input genomes is plotted at increasing levels of genomic complexity. Each data points shows the mean value across 100 000 runs. The results under each of the different weighting ratios, one-to-one ratio (red), same-as solution ratio (blue) and pseudo-randomized ratio (green) are plotted on the graph.

Variation in processing speed will also influence the variability of the data within the 100 000 runs.

In general, it appears that Genolve takes longer to complete execution under a one-to-one weighting ratio. At a genomic complexity level of 8|9, however, the time taken under a same-as-

solution ratio surpasses the one-to-one ratio. This corresponds to what was observed in Section 3.3.1. when investigating the average number of solutions per solution set. In Fig 28, the average number of solutions per solution set at a genomic complexity level of 8|9 under the same-as-solution ratio, was higher than under the one-to-one ratio. A higher number of solutions within a solution set would require more time for network traversal to identify all the various solutions.

Similarly, when comparing Figures 28 and 35, the average number of solutions within a solution set under the pseudo-randomized ratio in Fig 28 is lower than under the other two ratios. This corresponds with the lower average runtime of Genolve under the pseudo-randomized ratio in Fig 34 compared to the results under the other two ratios.

Each of the data points in Fig 35 represents a mean value across 100 000 runs. The spread of the values constituting these averages is shown as box-and-whisker plots in Fig 36.1 – 36.6, below.

The often-vast number of outliers, defined as exceeding $Q3 + 1.5 \times IQR$ or falling below $Q1 - 1.5 \times IQR$, where $Q1$ is the first quartile, $Q3$ is the third quartile and IQR is the interquartile range ($Q3 - Q1$) makes it difficult to see the box-and-whisker component of the plots. To resolve this, for each of the figures, the entire plot is shown in a) and the enlarged box-and-whisker component is shown in b).

As was the case with the variations within the number of solutions per solution set, the variation in execution time for different runs of Genolve is relatively large. This was not unexpected, since, as was previously mentioned, some evolutionary scenarios between input genome pairs are significantly more complex than others. Increased complexity results in an increase in the size of the network of solutions that Genolve needs to compute and traverse, which has a negative impact on the runtime of Genolve.

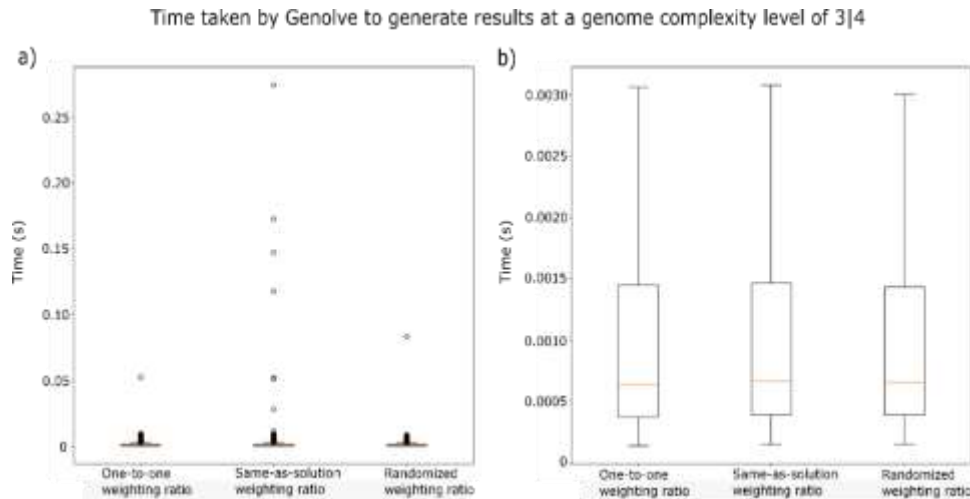


Fig 36.1. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 3|4 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

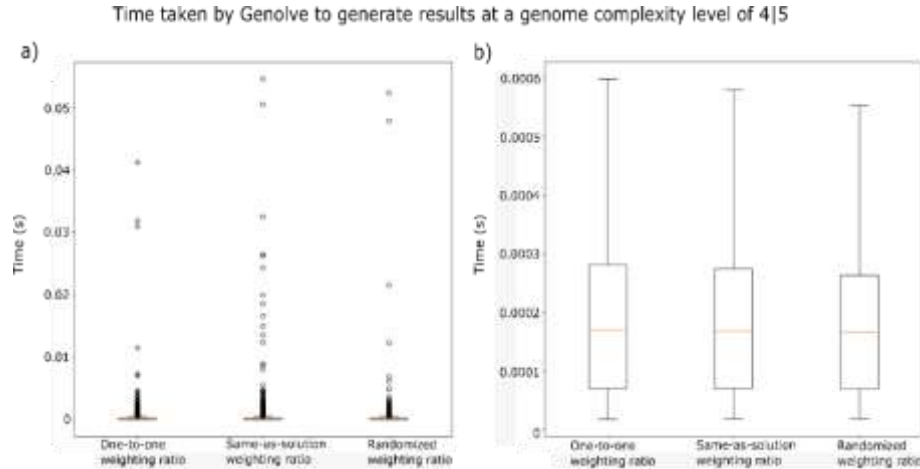


Fig 36.2. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 4|5 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

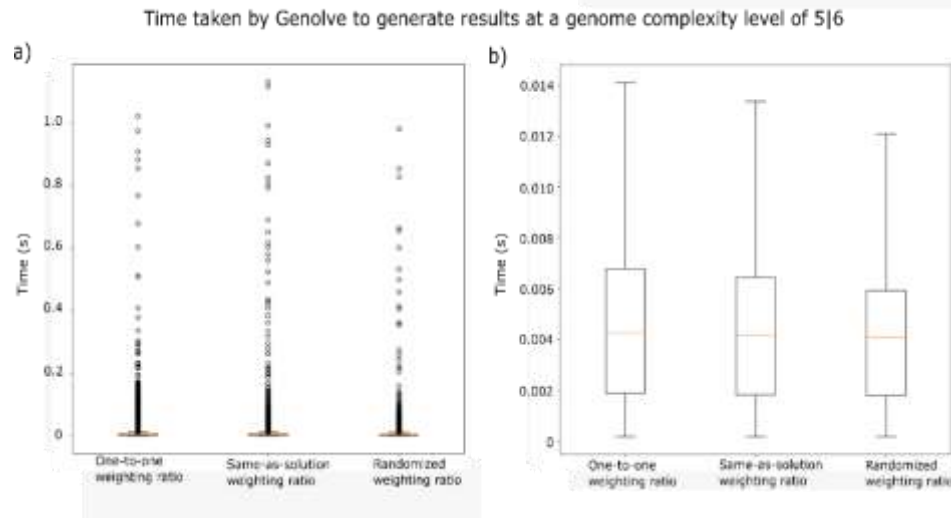


Fig 36.3. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 5/6 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

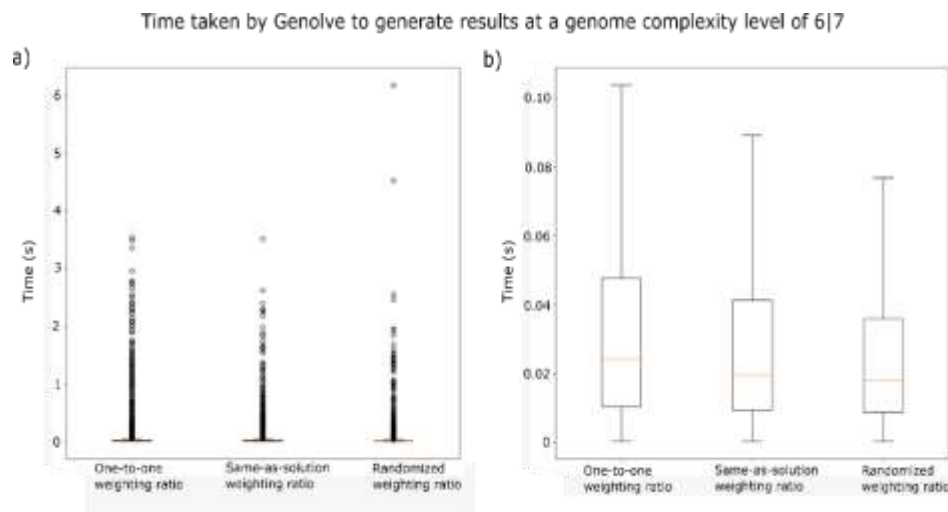


Fig 36.4. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 6/7 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

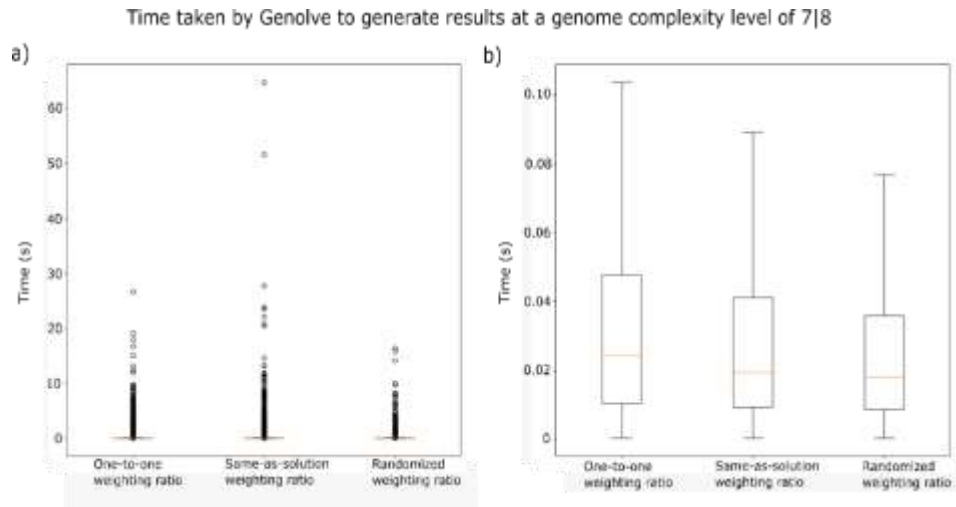


Fig 36.5. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 7|8 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

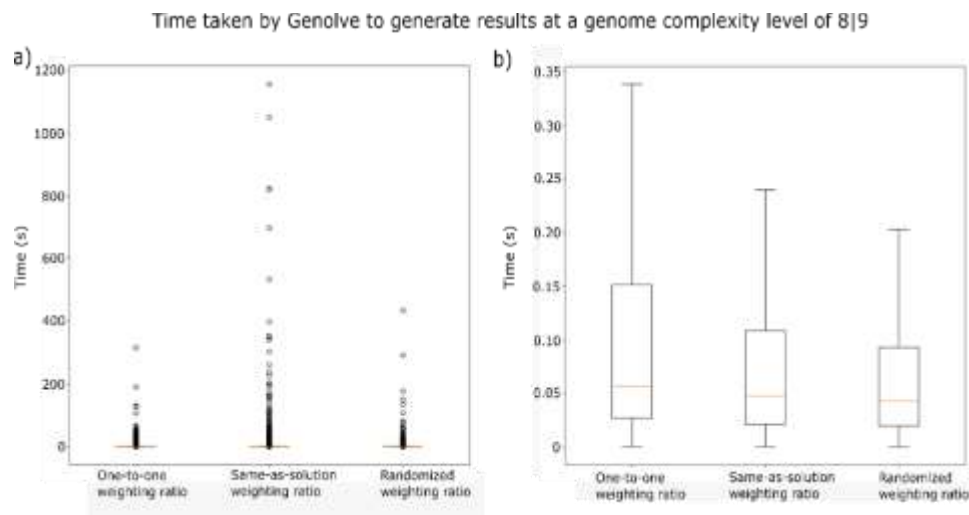


Fig 36.6. A box-and-whisker plot of the time Genolve takes to generate a solution set for a single pair of input genomes under each of the three different weighting ratios, one-to-one, same-as-solution and pseudo-randomized at a genomic complexity level of 8|9 is shown. **a)** outliers are included in the plot **b)** outliers are excluded to allow for a clearer visual of the box-and-whisker component of the graph. The rectangles indicate the range from the first to the third quartile in each data set. Whiskers indicate a position that exceeds or falls below the third quartile ($Q3$) and first quartile ($Q1$) respectively by $1.5 \times$ the interquartile range ($Q3-Q1$). The orange line indicates the median of each data set.

3.6.1.1. Variability in processing power

However, as has been mentioned, another factor that may influence this variation is the variation in the availability of computing power of the HPC at any given moment. There are often additional jobs being submitted or executing jobs that finish, whilst Genolve is executing. This constant initiation and termination of jobs will result in a fluctuation in processing power dedicated to Genolve. To investigate this, I analysed the same pair of input genomes 1 000 times in a single job on the HPC. All 1 000 of these runs would compute and traverse the same network of solutions. No variation between the 1 000 runs was thus expected, and any variation could be attributed to fluctuations of processing ability of the HPC. The results are plotted in Fig 37, below.

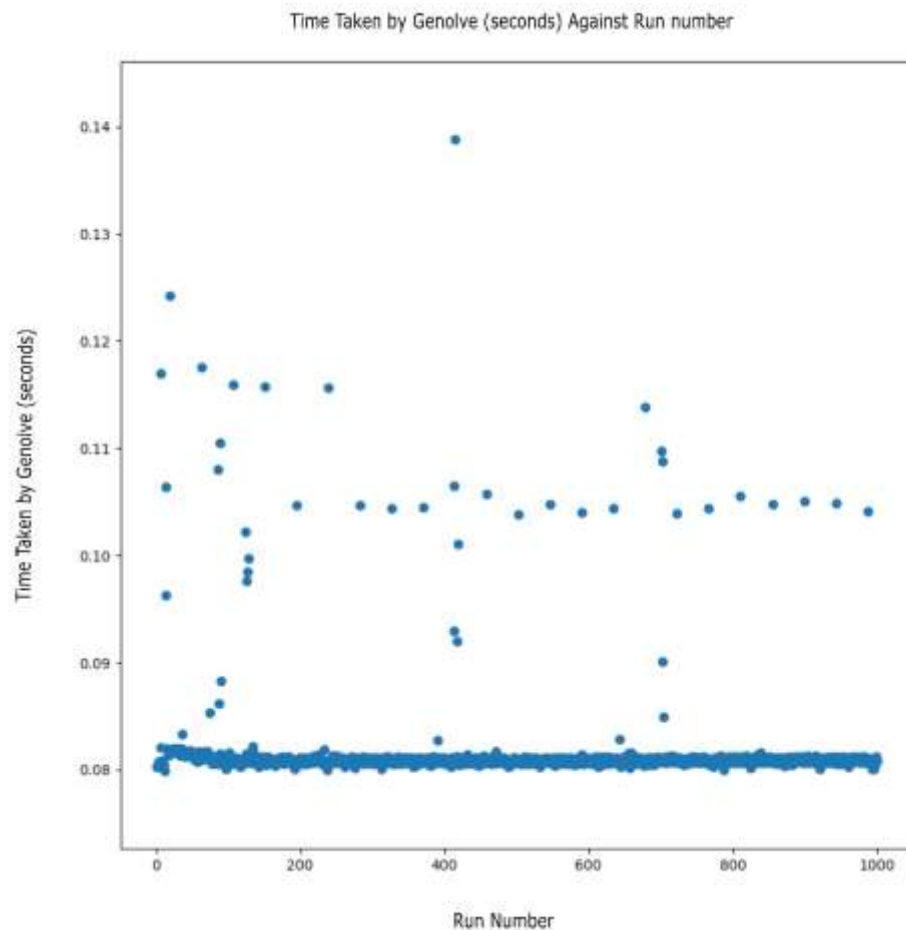


Fig 37. A scatter plot of the time (seconds) Genolve takes to solve the same input genome pair for 1000 runs. The average time taken by Genolve = 0.08187 ± 0.00528 seconds.

If no variation was present, all the data points would be expected to lie on a single horizontal line. There is a clear horizontal line just above 80 milliseconds. There are also, however, a number of data points which lie far above the horizontal line. These may be a result of a new job being submitted to the HPC at the time of execution of those runs. The average runtime of Genolve across the 1000 runs was 0.08187 ± 0.00528 seconds.

3.6.1.2. Big Oh Notation

To get a more concrete idea of the runtime of an algorithm in terms of scalability, it is often useful to calculate the Big Oh time complexity expression of the algorithm. This notation always gives the worst-case scenario and shows how the runtime of an algorithm scales with the size of the input.

This subsection will give the Big Oh notation of the Genolve algorithm, show how it was calculated, and why the worst case is not expected to occur frequently.

For this purpose, the input size, n , is the number of non-final state adjacencies present in the source genome. If the source genome is completely disordered relative to the target genome then:

$$n = \text{number of sequence blocks} - 1$$

I will show that the time complexity of Genolve is $O(n!)$.

In the worst-case scenario, every DCJ operation generates only one final-state adjacency. This means that each path of transformation of source into target would take the same number of steps as there are non-final state adjacencies, namely n steps.

Another assumption made under a worst-case scenario is that all rearrangements are independent. I.e. that no rearrangement be restricted to occur only after the occurrence of some other rearrangement. This means that the full set of solutions will contain every permutation of the different set of rearrangements able to achieve full transformation of the source genome into the target genome.

Theoretically, it is also possible that no paths in the network of solutions converge (an example of a network containing converging paths can be found in Fig 17 in Section 2.3.2.), and that the path

from source to target genome has to be calculated separately for each path, so that for n non-final state adjacencies, there are n possible rearrangement operations that will initiate n different paths.

After execution of a single operation, each path will have $n-1$ non-final state adjacencies and thus $n-1$ possible operations (*i.e.*, different paths springing from it). After execution of a second rearrangement, there will be $n-2$ operations possible at each path, splitting each path into another $n-2$ branches, and so on. An illustration of this is shown in Fig 38 below. The runtime up until a particular level of the network of solutions (going from the source genome down to the target genome) is shown in red on the figure's left.

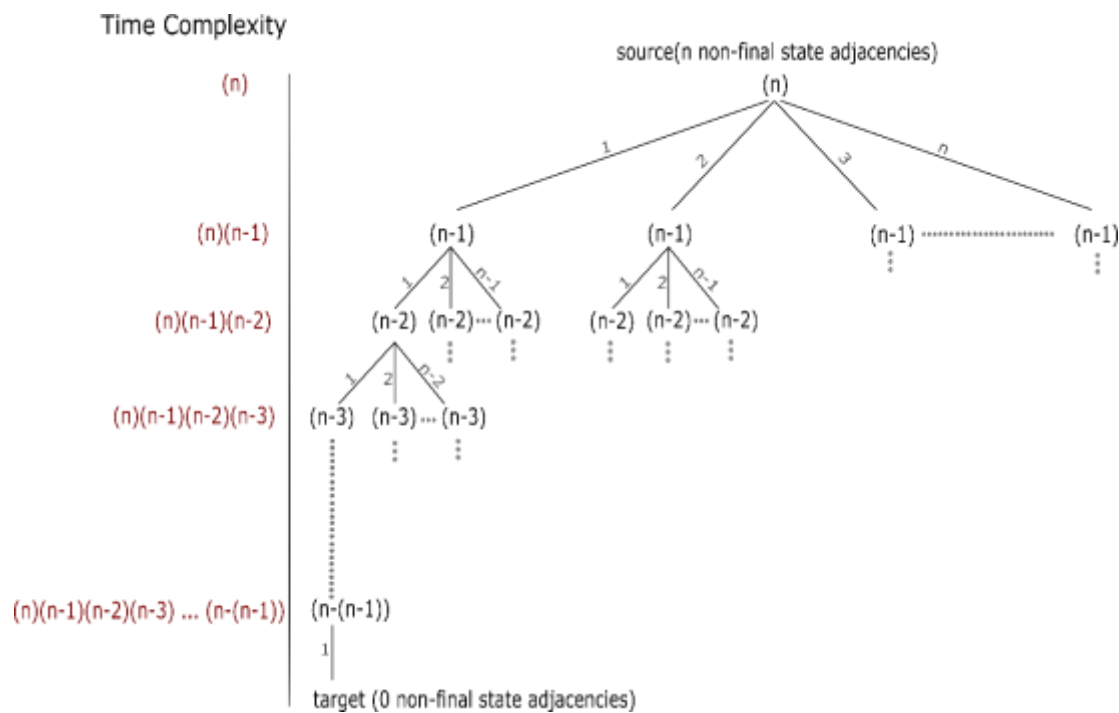


Fig 38. An illustration of how the worst-case time complexity for the Genolve algorithm can be calculated. The symbol n represents the number of non-final-state adjacencies present in the source genome. The tree structure shows how the number of branches splitting from any one point in a path relates to n . The runtime up to a particular level of the network of solutions (going from the source genome down to the target genome) is shown in red on the left.

The worst-case time complexity of the algorithm is thus:

$$\text{time complexity} = (n)(n-1)(n-2)(n-3) \dots 1 = n!$$

A time complexity of $O(n!)$ is not particularly desirable. For our purpose, it is, however, unavoidable since the function of Genolve is to identify *all* most parsimonious rearrangement scenarios that will result in the transformation of one genome into another. Genolve, therefore, has to calculate all the permutations of all the different rearrangement events that may occur together.

It is important to mention, however, that a time complexity of $O(n!)$ will seldom be realized, as the two assumptions made for the worst-case scenario, namely that only a single non-final state adjacency is resolved per DCJ operation and that there is no convergence in the network of solutions, rarely hold true in practice.

3.6.2 Space complexity and memory usage of Genolve.

3.6.2.1 Space complexity

Similar to the worst-case time complexity, the worst-case space complexity is dependent on the number of intermediate genomes that need to be calculated and added to both the dictionary of intermediates and as nodes in the network of solutions. The worst-case space complexity is thus also $O(n!)$ where n is the number of non-final state adjacencies present in the source genome.

Again, as with time complexity in the previous section, the worst case will rarely be realized as it often happens that

- (i) DCJ operations create two final state adjacencies (not only one, which is the assumption under the worst case),
- (ii) not all rearrangements are independent and
- (iii) that solution paths in the network of solutions converge.

3.6.2.2 Memory usage

For a more tangible investigation of the memory usage of Genolve, it was applied to ten different input genome pairs for each of the following levels of genomic complexity: 15|16, 16|17, 17|18, 18|19, 19|20.

The small number of runs per level of genomic complexity made it feasible to use input genomes of higher complexity levels than has been used thus far, giving valuable information regarding the

usage of the tool. As was seen in the section on the run time of Genolve, the higher the level of genomic complexity, the longer Genolve will take to generate an output. Performing 100 000 runs at higher levels of genomic complexity was therefore not feasible.

However, it is extremely important to note that such a small sample size means that the data cannot be used to draw any rigorous conclusions from the dataset as a whole and I will not attempt to do so. This data aims, not to make comparisons, nor to come to any broad conclusions, but merely to investigate and discuss individual cases.

The memory required to calculate the solutions of each of the input genome pairs were recorded, as was the time taken for the calculation. The results are displayed in the table below.

Table 3. The memory usage (kb) and run time (h:m:s) for each of the ten runs at each of the five different levels of genomic complexity.

	Genome complexity: 16 15		Genome complexity: 17 16		Genome complexity: 18 17	
Run number	Memory usage (kb)	Time (h:m:s)	Memory usage (kb)	Time (h:m:s)	Memory usage (kb)	Time (h:m:s)
1	140866964	4:39:40	87272384	1:36:23	3069996	0:03:04
2	2300420	0:04:41	48338236	0:56:31	921800	0:01:49
3	229884	1:02	1132592	0:01:21	208396456	2:29:34
4	71968	0:00:35	258804	0:01:31	69915592	1:32:15
5	418948	0:01:34	69052	0:00:16	2082032	0:05:09
6	69672	0:00:15	0	0:00:05	2082032	0:05:09
7	162660944	1:29:13	2429736	0:03:48	210376	0:00:49
8	667776	0:00:51	31720	0:00:09	3937040	0:04:29
9	5349900	0:06:19	433540	0:00:39	463116	0:01:21
10	84312	0:00:16	2713308	0:03:48	65899864	1:17:12

	Genome complexity: 19 18		Genome complexity: 20 19	
Run number	Memory usage (kb)	Time (h:m:s)	Memory usage (kb)	Time (h:m:s)
1	1497624	0:01:48	60525240	1:41:29
2	159635224	3:12:22	258365196	3:51:11
3	135162816	2:29:54	883824	0:07:58
4	2456112	0:06:30	8412704	0:39:13
5	647060	0:01:54	212422172	2:00:35
6	19307432	0:23:21	263140	0:05:32
7	16680556	0:42:45	1033172	0:05:35
8	836508	0:09:52	41483988	1:07:18
9	1540516	0:09:46	152228	0:01:36
10	2838336	0:06:18	514982448	8:38:13

As has been mentioned previously in Sections 3.3.1. and 3.6.1., respectively, a large amount of variation in solution set size (and by extension memory usage) and time taken to generate the solution set exists between different input genome pairs even under the same level of genomic complexity. This is the result of the varying levels of complexity of the network of solutions that needs to be calculated. Fewer independent rearrangements and a high level of path convergence would result in smaller, less complex networks. A clear, direct correlation between the amount of memory used and the tool's run time is visible in the table.

The calculations of time and space complexity showed that both the runtime and memory usage of the algorithm had a worst-case complexity of $O(n!)$ where n is the number of non-final state adjacencies in the source genome. It was also mentioned that both the runtime and memory usage is rarely expected to reach these worst cases, for the same reasons – DCJ operations often create

two final state adjacencies, DCJ operations are not always independent and there is often the convergence of paths in the network of solutions. The same factors that would result in a lower runtime will thus also lead to lower memory usage. Runs with lower memory usage are thus expected to have lower runtimes than those with a high memory usage – this correlation is visible in the data.

As such the memory usage of Genolve is expected to increase with the runtime (Section 3.6.2) (given that the variability in run time caused by other scripts running on a node is not too big). As such, as with the runtime of the tool the memory usage is also highly variable and dependent on the complexity of the network of solutions – something that cannot be determined prior to running the tool.

Looking closer at individual runs, it quickly becomes apparent that the memory usage of run 6 under the genomic complexity level of 16|17 is less than 1kb. In contrast, run 1, under the same level of genomic complexity, has a memory usage of 87272384. Not only do these memory usages correlate with the run times of the runs, 5 seconds and 1 hour, 36 minutes and 23 second respectively, but upon closer inspection of the output of these runs, it becomes apparent there is also a correlation with the number of solutions within the solution sets and the number of operations per solution, in other words with the complexities of the networks of solutions.

The results summary and the first solution within the solution set is shown below for run 1 and 6 under the genomic complexity level 16|17 in Fig 39a and 39b, respectively.

a) #####

Source Genome: [[13, 1, -12, -6, 2, 3, 4, -15, -14, -10], [-17], [-11, -5, 9, 16], [-7], [8]]
 Target Genome: [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12], [13], [14, 15, 16, 17]]

Number of most parsimonious solutions: 5544000

Average number of operations per solution: 11.0

Average number of each operation per solution:
 Inversions: 0.0 Transpositions type 1: 2.0 Transpositions type 2: 0.0 Balanced translocations: 1.0 Unbalanced translocations: 1.0
 Fusions: 4.0 Fissions: 3.0

Solutions:

Solution number 1
 ([[7], [8], [10, 14, 15, -4, -3, -2, 6, 12, -1, -13], [-11, -5, 9, 16], [17]], ('none, this is the source genome', 'N/A'))
 ([[7], [8], [10, 14, 15, -4, -3, -2, -1, -13], [-11, -5, 9, 16], [17], ['o', 6, 12]], ('trp0', (((1.5, 12.5), (2, 6)), ((1.5, 2), (6, 12.5)))))
 ([[7], [8], [10, 14, 15, -4, -3, -2, -1, -13], [-11, -12, -6, -5, 9, 16], [17]], ('trp1', (((5.5, 11), (6, 12.5)), ((5.5, 6), (11, 12.5)))))
 ([[7], [8], [10, 14, 15, 9, 16], [-11, -12, -6, -5, -4, -3, -2, -1, -13], [17]], ('b_trl', (((4.5, 15.5), (5, 9)), ((4.5, 5), (9, 15.5)))))
 ([[7], [8], [10, 14, 15, 16], [-11, -12, -6, -5, -4, -3, -2, -1, -13], [17], ['o', 9]], ('trp0', (((9, 15.5), (9.5, 16)), ((9, 9.5), (15.5, 16)))))
 ([[7], [8, 9], [10, 14, 15, 16], [-11, -12, -6, -5, -4, -3, -2, -1, -13], [17]], ('trp1', (((9, 9.5), 8.5), ((8.5, 9), 9.5))))
 ([[1, 2, 3, 4, 5, 6, 12, 11], [7], [8, 9], [10, 14, 15, 16], [13], [17]], ('fis', ((1, 13.5), 1, 13.5)))
 ([[1, 2, 3, 4, 5, 6, 12, 11], [7], [8, 9], [10], [13], [14, 15, 16], [17]], ('fis', ((10.5, 14), 10.5, 14)))
 ([[1, 2, 3, 4, 5, 6, 12], [7], [8, 9], [10], [11], [13], [14, 15, 16], [17]], ('fis', ((11, 12.5), 11, 12.5)))
 ([[1, 2, 3, 4, 5, 6, 7], [8, 9], [10], [11], [12], [13], [14, 15, 16], [17]], ('u_trl', (((6.5, 12), 7), ((6.5, 7), 12))))
 ([[1, 2, 3, 4, 5, 6, 7, 8, 9], [10], [11], [12], [13], [14, 15, 16], [17]], ('fus', (7.5, 8, (7.5, 8))))
 ([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11], [12], [13], [14, 15, 16], [17]], ('fus', (9.5, 10, (9.5, 10))))
 ([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12], [13], [14, 15, 16], [17]], ('fus', (11.5, 12, (11.5, 12))))
 ([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12], [13], [14, 15, 16, 17]], ('fus', (16.5, 17, (16.5, 17))))

#####

b) #####

Source Genome: [[-17, 12, 9, 14, -11, -10, 1, 2, 3, 4, 5, 13, 6, 7], [-16, 8, -15]]
 Target Genome: [[1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11], [12, 13, 14, 15, 16, 17]]

Number of most parsimonious solutions: 900

Average number of operations per solution: 6.0

Average number of each operation per solution:
 Inversions: 0.48 Transpositions type 1: 3.0 Transpositions type 2: 0.0 Balanced translocations: 0.0 Unbalanced translocations: 1.52
 Fusions: 0.0 Fissions: 1.0

Solutions:

Solution number 1
 ([[-7, -6, -13, -5, -4, -3, -2, -1, 10, 11, -14, -9, -12, 17], [15, -8, 16]], ('none, this is the source genome', 'N/A'))
 ([[-7, -6, -5, -4, -3, -2, -1, 10, 11, -14, -9, -12, 17], [15, -8, 16], ['o', 13]], ('trp0', (((5.5, 13), (6, 13.5)), ((5.5, 6), (13, 13.5)))))
 ([[-7, -6, -5, -4, -3, -2, -1, 10, 11, -14, -9, -13, -12, 17], [15, -8, 16]], ('trp1', (((9, 12.5), (13, 13.5)), ((9, 13.5), (12.5, 13)))))
 ([[-7, -6, -5, -4, -3, -2, -1, 10, 11, -14, -13, -12, 17], [15, -8, 16], ['o', 9]], ('trp0', (((9, 13.5), (9.5, 14)), ((9, 9.5), (13.5, 14)))))
 ([[-7, -6, -5, -4, -3, -2, -1, 9, 10, 11, -14, -13, -12, 17], [15, -8, 16]], ('trp1', (((1, 10), (9, 9.5)), ((1, 9), (9.5, 10)))))
 ([[-7, -6, -5, -4, -3, -2, -1, 9, 10, 11, -14, -13, -12, 17], [15, 16], ['o', 8]], ('trp0', (((8, 16), (8.5, 15.5)), ((8, 8.5), (15.5, 16)))))
 ([[-8, -7, -6, -5, -4, -3, -2, -1, 9, 10, 11, -14, -13, -12, 17], [15, 16]], ('trp1', (((8, 8.5), 7.5), ((7.5, 8), 8.5))))
 ([[1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11, -14, -13, -12, 17], [15, 16]], ('fis', ((1, 9), 1, 9)))
 ([[1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11], [-16, -15, -14, -13, -12, 17]], ('u_trl', (((11.5, 14.5), 15), ((14.5, 15), 11.5))))
 ([[1, 2, 3, 4, 5, 6, 7, 8], [9, 10, 11], [12, 13, 14, 15, 16, 17]], ('inv', (((12, 17), 16.5), ((16.5, 17), 12))))

#####

Fig 39. The results summary and first solution with in the solution set for **a)** run 1 at the genomic complexity level of 16/17 and **b)** run 6 at the genomic complexity level of 16/17. Inv: inversion, trp0: circularization step of the transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion.

The solution set generated in run 1 consists of 5544000 solutions, each of which consists of 11 operations. In contrast, the input genome pair for run 6 required less rearrangement to sort the source genome into the target genome, and this run has a solution set size of 900 with only six operations per solution. The complexity of the network of solutions for run 1 was thus remarkably higher than for run 6.

Chapter 4: Testing Genolve on biological data

In the previous chapter, synthetic data was used to analyze and evaluate Genolve's performance. The purpose of this chapter is to investigate of Genolve's performance when applied to *biological data*.

The organization of the chapter will be as follows:

The first section describes the nature of the biological dataset selected. Section two gives a brief overview of the bioinformatics tool Mauve [62], which is used to generate lists of shared homologous regions between two genomes that serve as the input for Genolve.

The final section outlines the process of using Mauve and Genolve to generate the various rearrangement scenarios that describe the evolutionary relationship of a primitive and adapted strain of *Saccharomyces cerevisiae* successfully.

4.1. *Saccharomyces cerevisiae*

The budding yeast, *Saccharomyces cerevisiae*, more commonly known as baker's or brewer's yeast, has played an essential role in fermentative processes for more than ten thousand years [69], with the earliest evidence for its use in wine production dating back to the Neolithic era [70]. The yeast has also become an invaluable model organism for studying eukaryotic cell biology [71], being the first fully sequenced eukaryotic genome [72]. Research on *Saccharomyces cerevisiae* has led to numerous, notable advances of our understanding of living systems [71], some of which include a better understanding of the process of ageing [73], DNA damage response kinases [74], regulation of gene expression [75], transport across membranes [76], mitochondrial biology [77], signaling pathways [78], cell cycle regulation [79] and cell death [80].

Despite the fundamental role this species has played in scientific history, our knowledge of domesticated *Saccharomyces cerevisiae*'s evolutionary history remains sparse [81].

In 2012 Wang et al. [82] published a paper outlining a large scale field survey that pointed to the diversity of environments in which the species can be found – ranging from man-made environments to areas primarily devoid of human interference such as primeval forests.

Highly diverged wild lineages of the yeast, including the oldest lineages found to date, are located in China. This, in combination with a series of other studies [83–85], points to an out-of-China (and surrounding areas) origin for the *Saccharomyces* genus. For this reason, wild yeast populations from the region are invaluable when investigating the evolutionary history of the species.

The study by Duan et al. [81] outlines the invaluable analysis of 106 wild and 160 fermentation associated isolates of the species from a wide range of locations in China. The dataset includes fully sequence genomes of the oldest wild lineages from primeval forests and numerous domesticated lineages used in diverse fermentative process.

This type of data is beneficial for investigating the evolutionary history of primitive and domesticated yeast strains, and it was thus from this dataset that the two genomes to be analyzed by Genolve were selected.

4.2. Mauve

Mauve [62] is a tool used for genome comparison and alignment. In contrast to earlier methods of genome alignment, the tool is able to align multiple sequences in the presences of large scale genomic rearrangements, and in so doing, integrates traditional sequence alignment with the study of genome evolution [62].

Mauve's ability to align sequence regions in the presence of genomic rearrangements lies in identifying locally homologous regions between input sequences. These regions, called locally collinear blocks (LCBs) occur in a different order in one or more of the input genomes with respect to the rest and are used during the alignment of genomes [62]. These two series of genomic blocks, each with a unique number within a set, provide the series of numbers for the two genomes that serve as input to Genolve.

The protocol by Darling et al. [86] gives a comprehensive overview of the methodology behind and usage of Mauve.

4.3. The process

4.3.1 Generating the input files for Genolve

The two strains of *Saccharomyces cerevisiae*, selected from the dataset published by [81], were both isolated from the province of Guizhou in China. The wild type strain, FJSA40.2 (GenBank assembly accession: GCA_003275835.1), of the lineage CHN-X was sourced from the bark of a primeval forest in the Fanjing Mountain region. The domesticated strain, MTZ13.12 (GenBank assembly accession: GCA_003271185.1), was extracted from soil from a distillery and forms part of the Baiju lineage.

Mauve was used to identify the regions of homologous sequence (or LCBs) between the two genomes. Note that the minimum weight (i.e. length) of an LCB was set to 1300bp.

Fig 40 below shows the graphic depiction of the aligned genomes generated by Mauve. The different coloured blocks indicate shared homologous regions that occur in a different order in MTZ13.12 (bottom of figure) relative to the genome of FJSA40.2 (top of figure).

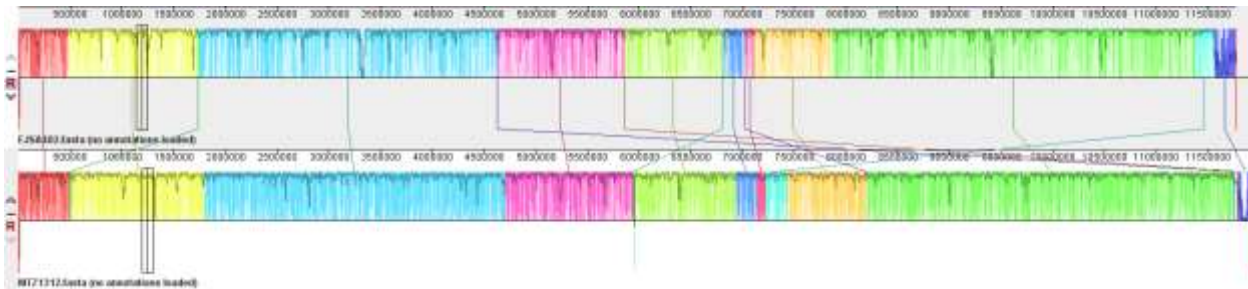


Fig 40. Alignment by the MauveAligner of the genomes of the strains FJSA40.2 (top) and MTZ13.12 (bottom). Locally colinear blocks are shown as coloured regions occurring in a different order in the two sequences.

In addition to an alignment file, Mauve can also, upon specification, output a permutation matrix. This matrix has a row for every input genome. Each row consists of a tab-separated list of integers representing the shared homologous regions between the genomes (i.e. the coloured blocks in the

graphic). The integers in the first row of the matrix are always consecutive, whilst those in the second are permuted, indicating the position of homologous sequence blocks in the two genomes.

The permutation matrix for the pair of selected *Saccharomyces cerevisiae* genomes is shown in Fig 41a below. Though this output of Mauve may appear to be in the appropriate format to input into Genolve, it is important to note that the permutation matrix gives no indication of the chromosomal localization of each of the respective colinear blocks. In addition to the permutation matrix, Mauve also generates a file with the base pair positions (start and end position) of each of the locally colinear blocks (Fig 41b). Using this data and mapping it to the data of the lengths (bp) of each of the chromosomes of the respective genomes, it is possible to deduce the chromosomal localization of each block.

a)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	2	1	3	5	-8	7	9	11	14	12	10	13	4	6

b)

#seq0_leftend	seq0_rightend	seq1_leftend	seq1_rightend
1923	475249	1	490399
475675	1721078	518297	1783348
1721079	1747957	490505	517939
1748747	4621508	1783835	4698695
4621876	4629125	11754425	11761675
4633884	5852358	4702924	5942547
5853393	5856586	11781368	11784561
5858290	6801334	5950011	6926282
6801335	6808369	-5942585	-5949641
6808370	7016898	6926583	7135379
7016899	7026674	8188488	8198268
7026678	7109134	7135682	7222043
7109801	7856787	7422682	8188103
7856796	11371615	8198269	11753881
11372307	11564542	7222055	7422681
11566079	11768451	11784738	11936443

Fig 41. **a)** permutation matrix generated by Mauve showing the order of the locally colinear blocks in the genome of strain FJSA40.2 (top) and MTZ13.12 (bottom). The negative sign in front of block 8 in the bottom row indicates the inversed orientation of the block relative to its homologous counterpart in the top row. **b)** the base pair locations of the locally colinear blocks for genomes FJSA40.2 (left-most two columns) and MTZ13.12 (right-most two columns), the negative signs in the eighth entries of the right-most two rows indicate the reverse orientation of the region.

Note that in Fig 41a, the 8th LCB in the bottom row (representing the LCBs of strain MTZ13.12) has a negative sign, indicating its inverse orientation with respect to its homologous counterpart in the genome of strain FJSA40.2. This inverse orientation can also be seen in the 8th row of columns three and four in Fig 41b where each of the entries is also negative.

The length of each chromosome comprising the genomes of *S. cerevisiae* strains FSAJ40.2 and MTZ13.12 were obtained from the NCBI entries of the genomes and are shown in Tables 4 and 5 below, respectively. The final column in each table shows the cumulative length of the chromosomes.

Table 4. Lengths of the respective chromosomes that make up the nuclear genome of the Saccharomyces cerevisiae strain FJSA40.2.

FJSA40.2		
Chromosome number	Length of Chromosome (bp)	Cumulative Length (bp)
1	196781	196781
2	758213	954994
3	308798	1263792
4	1480399	2744191
5	557950	3302141
6	276239	3578380
7	1054643	4633023
8	532851	5165874
9	419246	5585120
10	703833	6288953
11	664237	6953190
12	798680	7751870
13	894640	8646510
14	752992	9399502
15	1064856	10464358
16	907737	11372095

Table 5. Lengths of the respective chromosomes that make up the nuclear genome of the Saccharomyces cerevisiae strain MTZ13.12.

MTZ13.12		
Chromosome number	Length of Chromosome (bp)	Cumulative Length (bp)
1	197461	197461
2	807697	1005158

3	300409	1305567
4	1499782	2805349
5	573393	3378742
6	238885	3617627
7	1085282	4702909
8	531688	5234597
9	432894	5667491
10	733706	6401197
11	669692	7070889
12	1009713	8080602
13	923182	9003784
14	766411	9770195
15	1060667	10830862
16	923537	11754399

As was mentioned, using this chromosome length data in combination with the lengths of each of the individual LCBs in the respective genomes will enable the determination of the chromosomal localization of the LCBs. Tables 6 and 7 below, shows the start and end positions of each of the LCBs obtained from the output generated by MauveAligner (Fig 41b), the block length, as well as the colour it corresponds to on the graphic (Fig 40) for strains FJSA40.2 and MTZ13.12, respectively. Note that the data in Fig 41b for the genome of strain MTZ13.12 is not sorted by base pair position, but rather by the order in which the respective blocks are present in *FJSA40.2*. By reordering the rows by base pair positions (Table 7), the order of blocks correlates, with the order in which they appear in the graphic (Fig 40).

Table 6. Start and end positions of the locally colinear blocks (LCBs) identified by the MauveAligner for strain FJSA40.2. The length of each of the LCBs, as well as the colour of the LCB in the graphic generated by Mauve, is shown in the right-most columns of the table.

FJSA40.2			
Start position (bp)	End position (bp)	Length of LCB (bp)	Colour of LCB
1923	475249	473326	red
475675	1721078	1245403	yellow
1721079	1747957	26878	green
1748747	4621508	2872761	blue
4621876	4629125	7249	purple
4633884	5852358	1218474	pink

5853393	5856586	3193	brown
5858290	6801334	943044	lime
6801335	6808369	7034	turquoise
6808370	7016898	208528	dark blue
7016899	7026674	9775	violet
7026678	7109134	82456	peach
7109801	7856787	746986	bronze
7856796	11371615	3514819	forest green
11372307	11564542	192235	light blue
11566079	11768451	202372	royal blue

Table 7. Start and end positions of the locally colinear blocks (LCBs) identified by the MauveAligner for strain MTZ13.12. The length of each of the LCBs, as well as the colour of the LCB in the graphic generated by Mauve, is shown in the right-most columns of the table.

MTZ13.12			
Start position (bp)	End position (bp)	Length of LCB (bp)	Colour of LCB
1	490399	490398	red
490505	517939	27434	green
518297	1783348	1265051	yellow
1783835	4698695	2914860	blue
4702924	5942547	1239623	pink
-5942585	-5949641	7056	turquoise
5950011	6926282	976271	lime
6926583	7135379	208796	dark blue
7135682	7222043	86361	peach
7222055	7422681	200626	light blue
7422682	8188103	765421	bronze
8188488	8198268	9780	violet
8198269	11753881	3555612	forest green
11754425	11761675	7250	purple
11781368	11784561	3193	brown
11784738	11936443	151705	royal blue

Inspecting each of the start and end positions of the different blocks in the two genomes and comparing it to the length of the chromosomes of the respective genomes, allowed for the determination of the chromosomal localization of each block (see Tables 8 and 9 below).

Table 8. Chromosomes spanned by each of the locally colinear blocks (LCBs) identified by the MauveAligner in the genome of S. cerevisiae strain FJSA40.2. The LCBs are identifiable by the colours in which they appear on the graphic of the alignment generated by Mauve.

FJSA40.2	
Colour of LCB	Chromosome numbers
red	1, 2
yellow	2, 3, 4
green	4
blue	4, 5, 6, 7
purple	7
pink	8, 9
brown	9
lime	10, 11
turquoise	11
dark blue	11, 12
violet	12
peach	12
bronze	12, 13
forest green	13, 14, 15, 16
light blue	16
royal blue	16

Table 9. Chromosomes spanned by each of the locally colinear blocks (LCBs) identified by the MauveAligner in the genome of S. cerevisiae strain MTZ13.12. The LCBs are identifiable by the colours in which they appear on the graphic of the alignment generated by Mauve.

MTZ13.12	
Colour of LCB	Chromosome numbers
red	1, 2
green	2
yellow	2, 3, 4
blue	4, 5, 6, 7
pink	8, 9, 10
turquoise	10
lime	10, 11

dark blue	11, 12
peach	12
light blue	12
bronze	12, 13
violet	13
forest green	13, 14, 15, 16
purple	16
brown	16
royal blue	16

It should be noted that numerous colours or blocks span multiple chromosomes. Tables 8 and 9 can be reorganized so that the different colours *per chromosome* (instead of the different chromosomes *per colour*) becomes apparent (see Tables 10 and 11).

The colours on each of the respective chromosomes in Table 10 are assigned integer labels; these same labels are used in Table 11. Note that at chromosome 9, in strain FJSA40.2, there are only two coloured blocks comprising the chromosome (pink and brown), but there are three intergenic labels (13, 14, 15). This is because the pink region is confined to only two chromosomes in FJSA40.4 (chromosomes 8 and 9) but occurs over three chromosomes in MTZ13.12 (chromosomes 8, 9 and 10). This spread of a sequence block across more chromosomes in strain MTZ13.14 relative to strain FJSA40.2 is indicative of a fission event. Thus, it is necessary to break the pink region into three and not two separated regions to ensure that the fission event is identified.

The data in the final column of the tables, the intergenic values of the colours localized on each of the chromosomes, is in the correct format to serve as input for Genolve.

Table 10. The locally colinear blocks identified by the MauveAligner for contained on each of the chromosomes of the genome of the S. cerevisiae strain FJSA40.2. The colour in column two corresponds to the integer labels assigned to them in column 3.

FJSA40.2		
Chromosome number	LCB colours located on the chromosome	Genolve sequence blocks
1	red	1
2	red, yellow	2, 3
3	yellow	4
4	yellow, green, blue	5, 6, 7

5	blue	8
6	blue	9
7	blue, purple	10, 11
8	pink	12
9	pink, brown	13, 14*, 15
10	lime	16
11	lime, turquoise, dark blue	17, -18, 19
12	dark blue, violet, peach, bronze	20, 21, 22, 23
13	bronze, forest green	24, 25
14	forest green	26
15	forest green	27
16	forest green, light blue, royal blue	28, 29, 30

*Table 11. The locally colinear blocks identified by the MauveAligner for contained on each of the chromosomes of the genome of the *S. cerevisiae* strain MTZ13.12. The colour in column two corresponds to the intergenic labels in column 3.*

MTZ13.12		
Chromosome number	LCB colours located on the chromosome	Genolve sequence blocks
1	red	1
2	red, green, yellow	2, 6, 3
3	yellow	4
4	yellow, blue	5, 7
5	blue	8
6	blue	9
7	blue	10
8	pink	12
9	pink	13
10	pink, turquoise, lime	14, -18, 16
11	lime, dark blue	17, 19
12	dark blue, peach, light blue, bronze	20, 22, 29, 23
13	bronze, violet, forest green	24, 21, 25
14	forest green	26
15	forest green	27
16	forest green, purple, brown, royal blue	28, 11, 15, 30

It is interesting to note that prior to incorporating chromosomal localization of the LCBs, the respective genomes consisted of only 16 blocks each. After the incorporation of chromosomal localization, each of the genomes consists of 30 homologous blocks. This is as a result of a large

number of LCBs spanning multiple chromosomes. A unique intergenic label, therefore, needs to be assigned to each part of the LCB located on a different chromosome.

4.3.2. Running Genolve

Once the data was in the correct format, it became possible to run the data through Genolve (both input files for source and target genome can be found in the GitHub repository in which the code files for the Genolve tool is located under the file names FJSA402.txt and MTZ1312.txt). A one-to-one weighting ratio was used (i.e. each type of rearrangement was assigned the same weight).

The summary output as well the first solution generated by Genolve is shown in Fig 42.

There are 161280 different solutions for transforming the genome of the *Saccharomyces cerevisiae* strain MTZ12.13 into that of strain FJSA40.2, with transpositions being the most common type of rearrangement to occur. Interestingly, despite the inversed orientation of sequence block 18 in the source genome relative to the target genome, no inversions are present in any of the solutions. The reason for this will be the illegality of such an inversion under the DCJ model, i.e. at no point in the transformation process would the inversion of the -18 sequence block have resulted in the creation of at least one final state adjacency. In the target genome, sequence block 18 is present between sequence blocks 17 and 19, the final state adjacencies in which the extremities of sequence block 18 occurs is thus (17.5, 18) and (18.5, 19). At no point in the transformation process does the inversion of sequence block 18 results in one of the aforementioned adjacencies.

```

*****
Source Genome: [[1], [2, 6, 3], [4], [5, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 29, 23], [24, 21, 25], [26], [27], [28, 11, 15, 30]]
Target Genome: [[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13, 14, 15], [16], [17, 18, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]]

Number of most parsimonious solutions: 161280
Average number of operations per solution: 9.0

Average number of each operation per solution:
Inversions: 0.0 Transpositions type 1: 4.0 Transpositions type 2: 0.0 Balanced translocations: 1.675 Unbalanced translocations: 1.325 Fusions: 1.0 Fissions: 1.0

Solutions:

Solution number 1
[[1], [2, 6, 3], [4], [5, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 29, 23], [24, 21, 25], [26], [27], [28, 11, 15, 30]], ('none, this is the source genome', 'N/A')
[[1], [2, 3], [4], [5, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 29, 23], [24, 21, 25], [26], [27], [28, 11, 15, 30], ['o', 6]], ('trp0', (('2.5, 6), (3, 6.5)), (('2.5, 3), (6, 6.5))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 29, 23], [24, 21, 25], [26], [27], [28, 11, 15, 30]], ('trp1', (('5.5, 7), (6, 6.5)), (('5.5, 6), (6.5, 7))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 23], [24, 21, 25], [26], [27], [28, 11, 15, 30], ['o', 29]], ('trp0', (('22.5, 29), (23, 29.5)), (('22.5, 23), (29, 29.5))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 23], [24, 21, 25], [26], [27], [28, 29, 11, 15, 30]], ('trp1', (('11, 28.5), (29, 29.5)), (('11, 29.5), (28.5, 29))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 22, 23], [24, 25], [26], [27], [28, 29, 11, 15, 30], ['o', 21]], ('trp0', (('21, 24.5), (21.5, 25)), (('21, 21.5), (24.5, 25))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 11, 15, 30]], ('trp1', (('20.5, 22), (21, 21.5)), (('20.5, 21), (21.5, 22))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10], [12], [13], [14, -18, 16], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30], ['o', 11, 15]], ('trp0', (('11, 29.5), (15.5, 30)), (('11, 15.5), (29.5, 30))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11, 15], [12], [13], [14, -18, 16], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('trp1', (('11, 15.5), (10.5, 11), (15.5, 10.5))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13], [14, -18, 16], [15], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('trf', (('11.5, 19), (11.5, 15))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13, 14, -18, 16], [15], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('trf', (('13.5, 14), (13.5, 14))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13, 14, 15], [-16, 18], [17, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('u_trf', (('14.5, 18.5), (15), (14.5, 15), (18.5, 14.5))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13, 14, 15], [-16, 19], [17, 18], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('b_trf', (('16, 18), (17.5, 19), (16, 19), (17.5, 18))))
[[1], [2, 3], [4], [5, 6, 7], [8], [9], [10, 11], [12], [13, 14, 15], [16], [17, 18, 19], [20, 21, 22, 23], [24, 25], [26], [27], [28, 29, 30]], ('u_trf', (('16, 19), (18.5), (18.5, 19), (16, 19))))
*****

```

Fig 42. Results summary and first solution within the solution set generated by Genolve for an input genome pair of strain FJSA40.2 (target genome) and MTZ13.12 (source genome) of the yeast species *Saccharomyces cerevisiae*. Inv: inversion, trp0: circularization step of the transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion.

Genolve identified 161280 different plausible series of rearrangement events that could describe the evolution of the *S. cerevisiae* strain MTZ12.13 into that of strain FJSA40.2. Each solution consisted of 9 rearrangement events, with the average solution consisting of four transpositions, 1.675 balanced translocations, 1.325 unbalanced translocations, a chromosome fission and a chromosome fusion.

Prior to discussing the implications of these results, I would like to draw the reader's attention to the input genome pair. The source genome contains seven single sequence block chromosomes that are present in the target genome as well. Because these sequence blocks are already in their final state none of the rearrangements used to transform the source genome into the target genome acts upon them. As mentioned in previous sections, due to the algorithm's nature, a lower number of sequence blocks will increase the runtime. Doing away with the seven sequence blocks, by condensing the two genomes (see Fig 43), should lead to a faster runtime without affecting the number of solutions, the number of operations per solution, or the average number of the different types of operations per solution.

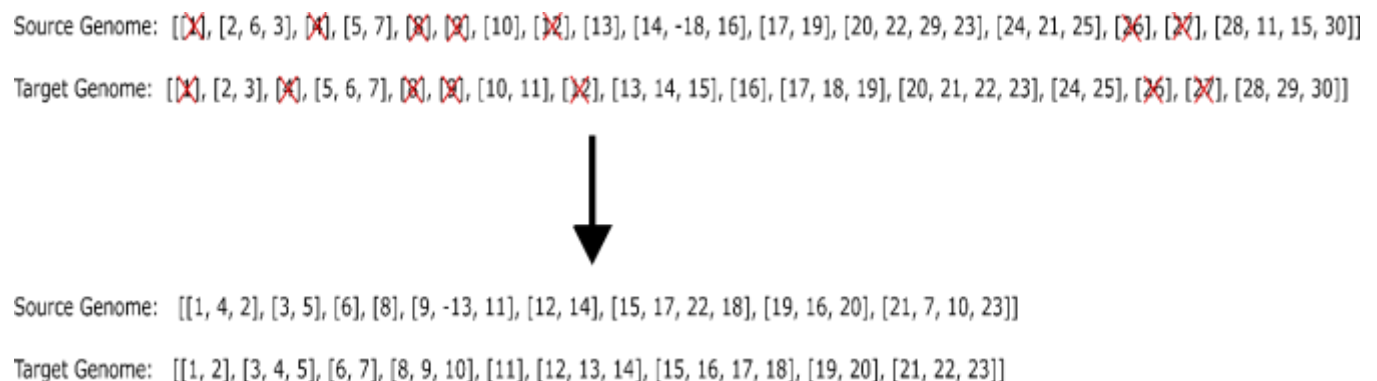


Fig 43. Illustration of how a source-target genome pair, in which the number of final-state adjacencies in the source genome is non-zero, can be condensed into genome with few sequence blocks by eliminating those sequence blocks that are already in their final states. The sequence

blocks to be eliminated in the top genomes are crossed out in red. At the bottom of the figure, the condensed genomes, with the necessary sequence blocks removed, is shown. The remaining sequence blocks are renumbered so that the target genome consists of consecutive integers.

Instead of an input genome pair consisting of 30 “genes” confined to 15 chromosomes, the condensed input genome pair consists of only 23 “genes” making up nine chromosomes. The results summary and first solution generated by Genolve for the condensed input genome pair is shown in Fig 44.

As expected, there is no difference in the number of solutions, average number of operations per solution not in the average number of each operation per solution. There was however a difference in the run time of Genolve for the uncondensed pair of input genomes compared to the condensed version. The runtime of Genolve for the uncondensed genomes was 256.5seconds, while the condensed version of the genomes ran in only 214.2 seconds.

Note, however, that the full difference in runtime is not necessarily attributable only to differences in the running of Genolve but also to differences brought about by the processing speed of the CPU on which Genolve is run.

```

*****
Source Genome: [[1, 4, 2], [3, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 22, 18], [19, 16, 20], [21, 7, 10, 23]]
Target Genome: [[1, 2], [3, 4, 5], [6, 7], [8, 9, 10], [11], [12, 13, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]]

Number of most parsimonious solutions: 161280

Average number of operations per solution: 9.0

Average number of each operation per solution:
Inversions: 0.0 Transpositions type 1: 4.0 Transpositions type 2: 0.0 Balanced translocations: 1.675 Unbalanced translocations: 1.325 Fusions: 1.0 Fissions: 1.0

Solutions:

Solution number 1
[[[1, 4, 2], [3, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 22, 18], [19, 16, 20], [21, 7, 10, 23]], ('none, this is the source genome', 'N/A')]
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 22, 18], [19, 16, 20], [21, 7, 10, 23]], ['o', 4]], ('trp0', (((1, 5, 4), (2, 4, 5)), ((1, 5, 2), (4, 4, 5))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 22, 18], [19, 16, 20], [21, 7, 10, 23]], ('trp1', (((3, 5, 5), (4, 4, 5)), ((3, 5, 4), (4, 5, 5))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 18], [19, 16, 20], [21, 7, 10, 23]], ['o', 22]], ('trp0', (((1, 7, 5, 22), (18, 22, 5)), ((1, 7, 5, 18), (22, 22, 5))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 18], [19, 16, 20], [21, 22, 7, 10, 23]], ('trp1', ((7, 21, 5), (22, 22, 5)), ((7, 22, 5), (21, 5, 22))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 17, 18], [19, 20], [21, 22, 7, 10, 23]], ['o', 16]], ('trp0', (((1, 6, 19, 5), (16, 5, 20)), ((1, 6, 16, 5), (19, 5, 20))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 7, 10, 23]], ('trp1', (((1, 5, 5, 17), (16, 16, 5)), ((1, 5, 5, 16), (16, 5, 17))))
[[[1, 2], [3, 4, 5], [6], [8], [9, -13, 11], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ['o', 7, 10]], ('trp0', (((7, 22, 5), (10, 5, 23)), ((7, 10, 5), (22, 5, 23))))
[[[1, 2], [3, 4, 5], [6, 7, 10], [8], [9, -13, 11], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('trp1', (((7, 10, 5), 6, 5), ((6, 5, 7), (10, 5))))
[[[1, 2], [3, 4, 5], [6, 7, 10], [9, -13, 11], [10], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('fus', ((7, 5, 10), 7, 5, 10))]
[[[1, 2], [3, 4, 5], [6, 7], [8, 9, -13, 11], [10], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('fus', (8, 5, 9, (8, 5, 9))))
[[[1, 2], [3, 4, 5], [6, 7], [8, 9, 10], [-11, 13], [12, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('u_tri', (((9, 5, 13, 5), 10), ((9, 5, 10), (13, 5))))
[[[1, 2], [3, 4, 5], [6, 7], [8, 9, 10], [-11, 14], [12, 13], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('b_tri', (((1, 13), (12, 5, 14)), ((1, 14), (12, 5, 13))))
[[[1, 2], [3, 4, 5], [6, 7], [8, 9, 10], [11], [12, 13, 14], [15, 16, 17, 18], [19, 20], [21, 22, 23]], ('u_tri', (((1, 14), (13, 5), (13, 5, 14), 11))))
*****

```

Fig 44. Results summary and first solution within the solution set generated by Genolve for the condensed input genome pair of strain FJSA40.2 (target genome) and MTZ13.12 (source genome) of the yeast species *Saccharomyces cerevisiae*. Inv: inversion, trp0: circularization step of the

transposition, trp1: reinsertion step of the transposition, b_trl: balanced translocation, u_trl: unbalanced translocation, fis: fission, fus: fusion.

Notice that despite the existence of an inversely orientated sequence block in the source genome (sequence block [-18] in the uncondensed genomes and sequence block [-13] in the condensed genomes), no inversions are present in the solutions generated by Genolve.

By inspecting the input genomes in Fig 44, the reason for this becomes apparent. The sequence block [-13] in the source genome is present on the chromosome [9, -13, 11]. The inversion of this sequence block to create the chromosome [9, 13, 11] does not contribute to the source genome being any more similar to the target genome. In contrast, if the sequence block [-13] is *transposed* to the chromosome [12, 14] (as is the case in the solutions shown in Fig 44) so that it becomes [12, 13, 14], a step has been taken towards the transformation of the source genome into the target genome as the transposition results in the creation of a chromosome already present in the target genome.

Unlike other tools, Genolve generates a full set of solutions for an input genome pair. In addition, Genolve also generates a summary of various metrics describing the solutions set. These summary metrics include the number of most parsimonious rearrangement scenarios, the number of rearrangements per solutions and the average number of each type of rearrangement per solution across the solution set (see Fig 44). Even when the solution set is too large for investigating each individual solution, valuable information about the solutions making up the solution set can be deduced by analyzing these summary metrics.

There are 161280 most parsimonious solutions, consisting of 9 rearrangements each. If the nine rearrangements were all independent (*i.e.*, no rearrangement had to precede any other rearrangement), the expected number of most parsimonious solutions would be all the permutations of the nine rearrangements. This equates to $9! = 362800$ different solutions. The fact that the number of solutions is considerably less than 362800 indicates that dependent rearrangements does exist within each of the solutions.

The number of solutions and number of rearrangements per solution metrics that Genolve output thus gives information on whether or not the solutions contain dependent rearrangements.

Another list of metrics included in the output of Genolve is the average number of each *type* of rearrangement per solutions. Considering Fig 44, it is apparent that each solution contains four transpositions, one fission and one fusion. The other three rearrangements consist of an average of 1.675 balanced and 1.325 unbalanced translocations.

The number of balanced and unbalanced translocations present in a solution will not be the same across all the solutions. It is, however, possible to determine how many of the 161280 solutions will contain a certain number of balanced translocations and unbalanced translocations, respectively, as I illustrate below.

Table 12 below show that, given a total number of 161280 solutions with three translocations per solution made up of an average of 1.675 balanced translocations and 1.325 unbalanced translocations, the total number of balanced and unbalanced translocations across all solutions is 270144 and 213696 respectively.

Table 12. Calculation of the total number of balanced and unbalanced translocations present across all solutions found by Genolve.

Number of solutions:		161280
Number of translocations per solution:		3
Average number of balanced translocations per solution:		1.675
Average number of unbalanced translocations per solution:		1.325
Total number of balanced translocations:	$1.675 \times 161280 =$	270144
Total number of unbalanced translocations:	$1.325 \times 161280 =$	213696
Total number of translocations	$270144 + 213696 =$	483840

This data can be used to determine how many balanced and unbalanced translocations are present in certain portions of the solution set. For example, it may be possible that x% of the solutions contains one balanced and two unbalanced translocations, y% of the solutions contains two balanced and one unbalanced translocation and the final z% of the solutions contains three balanced and no unbalanced translocations.

Hypotheses describing the number of balanced and unbalanced translocations present in different portions of the 161280 solutions can be tested by calculating the total number of balanced and unbalanced translocations present across all solutions under the hypothesized proportions and comparing them to the values calculated in Table 12 above. If the values are not equivalent, the hypothesis can be rejected.

To illustrate this, I will investigate the different hypothesized scenarios

- (i) **Scenario 1:** *half of the solutions within the solution set contain two balanced translocations and one unbalanced translocation and the other half of the solutions one balanced and two unbalanced translocations.*

The calculations for Scenario 1 are shown in Table 13 below. Under this hypothesized scenario, the number of balanced translocations will total 241920, as will the number of unbalanced translocations. These values do not equate to the correct values determined in Table 12. The scenario is thus an invalid explanation for the types of translocations contained in different portions of the solutions within the solution set.

Table 13. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 1.

Scenario 1: Half of the solutions contain two balanced translocations and one unbalanced translocation and the other half contains one balanced translocation and two balanced translocations			
	Number of solutions	Number of balanced translocations	Number of unbalanced translocations
1/2 of the solutions:	$\frac{1}{2} \times 161280$ = 80640	2×80640 = 161280	1×80640 = 80640
1/2 of the solutions:	$\frac{1}{2} \times 161280$ = 80640	1×80640 = 80640	2×80640 = 161280
	161280	241920	241920
		Total number of balanced translocations:	241920
		Total number of unbalanced translocations:	241920
		Total number of translocations:	483840

Looking at the metrics for the different types of translocations, 1.675 and 1.325, it is clear that merely dividing the solution set into two equal portions as was done above would not yield the correct total number of each type of translocation across the whole solution set.

A better guess would be Scenario 2 described below:

- (ii) **Scenario 2:** *two-thirds of the solutions within the solution set contains two balanced translocations and one unbalanced translocation whilst the remaining third of the solutions contains one balanced and two unbalanced translocations.*

The results and calculations for this scenario are shown in Table 14 below. Although the number of balanced and unbalanced translocations calculated in Scenario 2 is closer to the correct values than in Scenario 1, the values are still not correct. Scenario 2, therefore, also fails to describe the types of translocation contained in various portions of the total solution set.

Table 14. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 2.

Scenario 2: Two-thirds of the solutions consist of two balanced translocations and one unbalanced translocation and the other third of the solutions contains one balanced and two unbalanced translocations			
	Number of solutions	Number of balanced translocations	Number of unbalanced translocations
2/3 of the solutions:	$2/3 \times 161280 = 107520$	$2 \times 107520 = 215040$	$1 \times 107520 = 107520$
1/3 of the solutions:	$1/3 \times 161280 = 53760$	$1 \times 53760 = 53760$	$2 \times 53760 = 107520$
	161280	268800	215040
Total number of balanced translocations:			268800
Total number of unbalanced translocations:			215040
Total number of translocations:			483840

The final, and as we will see correct, scenario is the one described below

- (iii) **Scenario 3:** *There are 56448 more balanced translocations than there are unbalanced translocations. This portion of the total number of balanced translocations is present in*

solutions containing three balanced translocations and zero unbalanced translocations, meaning $56448/3 = 18816$ solutions within the solution set contains three balanced translocations. Of the remaining 142464 solutions, one half contains two balanced translocations and one unbalanced translocation and the other half, one balanced and two unbalanced translocations.

The calculations in Table 15 shows that this scenario gives the correct number of total balanced and unbalanced translocations across the entire solution set.

Table 15. Calculation of the number of balanced and unbalanced translocation present across all solutions under the hypothesis described in scenario 3.

	Number of solutions	Number of balanced translocations	Number of unbalanced translocations
Difference between the # of balanced and unbalanced translocations / 3:	$(270144 - 18816) / 3 = 213696 / 3 = 71232$	$3 \times 18816 = 56448$	$0 \times 18816 = 0$
Half of the number of solution left after subtracting the 18816 solutions:	$1/2 \times (161280 - 71232) = 45024$	$2 \times 71232 = 142464$	$1 \times 71232 = 71232$
Half of the number of solution left after subtracting the 18816 solutions:	$1/2 \times (161280 - 71232) = 45024$	$1 \times 71232 = 71232$	$2 \times 71232 = 142464$
	161280	270144	213696
Total number of balanced translocations:			270144
Total number of unbalanced translocations:			213696
Total number of translocations:			483840

Identifying the correct scenario gives additional information on the solutions within the solutions set. In addition to knowing that each solution contains four transpositions, one fission, one fusion, 1.675 balanced and 1.325 unbalanced translocations, it has become clear that of the 161280 solutions, 18816 contains three balanced translocations, 71232 solutions contain two balanced

translocations, and one unbalanced translocation and 71232 solutions contains one balanced and two unbalanced translocations. Alternatively, we can say:

- 11.667% of the solutions consist of four transpositions, three balanced translocations, one fusion and one fission,
- 44.167% of the solutions consist of four transpositions, two balanced translocations, one unbalanced translocation, one fusion and one fission,
- 44.167% of the solutions consist of four transpositions, one balanced translocation, two unbalanced translocations, one fusion and one fission.

The insight into the presence of dependent rearrangements and the portion of solutions that consist of certain types of rearrangement events gives novel insight into the possible rearrangement scenarios that could describe the evolution of one genome into another. Even when the solution set itself that Genolve generates is too large to analyse, these metrics gives information that is not available when using other existing tools.

4.3.3 Comparison with GRIMM and UniMoG

As discussed in Section 1.5, the bioinformatic tools, GRIMM [63] and UniMoG [60], are used to estimate evolutionary distance between genomes (i.e. the number of operations required to transform the one into the other). These tools also give *one* genomic sorting scenario that would describe this transformation, as well as an estimation of the number of different sorting scenarios that would transform the source genome into the target genome.

The output of Genolve for the condensed input genome pair of the *S. cerevisiae* strains FJSA40.2 and MTZ13.12 was compared to that of GRIMM (Fig 45) and UniMoG (Fig 46) for the same input genome pair.

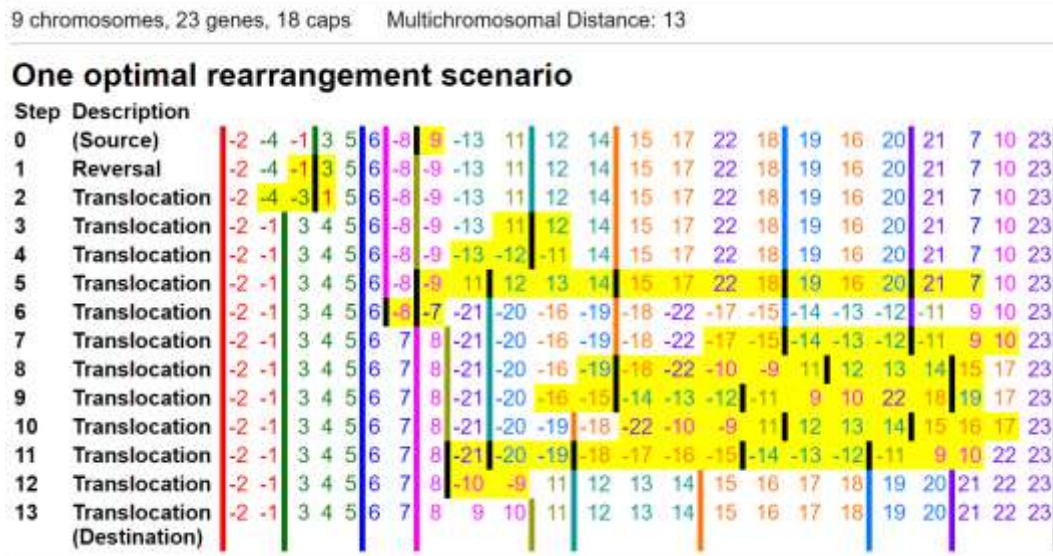


Fig 45. Sorting scenario generated by the GRIMM tool for the same input genome as in Fig 44. The colouring of the integers representing the sequence blocks correspond to the chromosomes on which they are located in the target genome. Sequence blocks that will be affected by rearrangement events are highlighted in yellow.

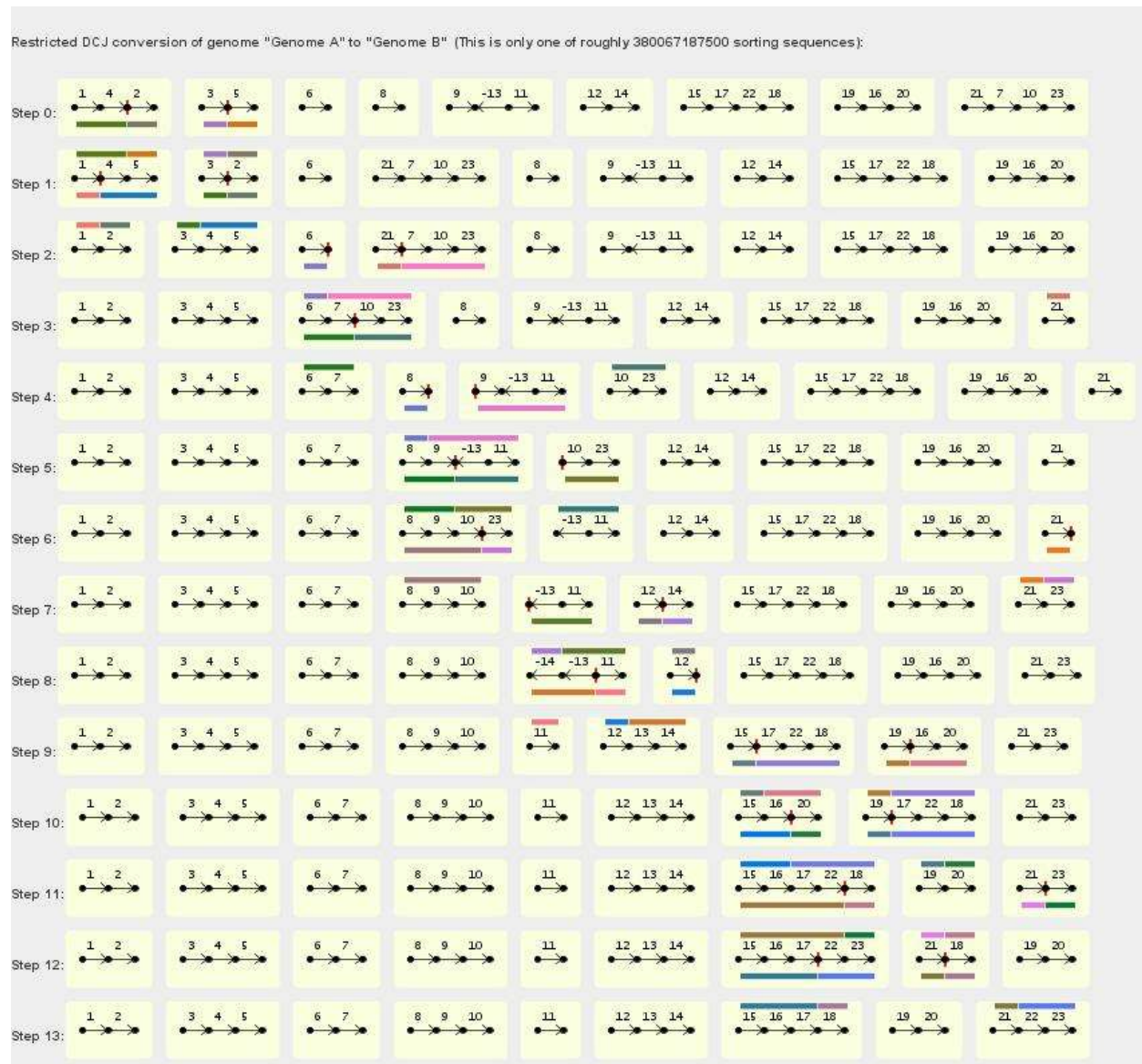


Fig 46. Sorting scenario generated by UniMoG, under the restricted DCJ model, for sorting the same source and target genomes as in Fig 44. The colouring under sequence blocks in one step corresponds to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.

Table 16 below summarizes of the similarities and differences between the three tools, each of which will be discussed separately.

Table 16. Comparison between the output of GRIMM, UniMoG and Genolve generated for the input genome pair of *S. cerevisiae* strains FJSA40.2 and MTZ13.12 in Fig 44.

	GRIMM	UniMoG	Genolve
Genomic sorting model used	HP model	rDCJ model*	Modified rDCJ model
Types of rearrangements used by the model	Inversions, translocations, fissions and fusions	Inversions, transpositions, translocations, fissions, fusions and block-interchanges	Inversions, transpositions, translocations, fissions, fusions and block-interchanges
Number of rearrangements per solution	13	13**	9
Estimated number of solutions	Does not give an estimation	380067187500	161280
Identification of rearrangements	Yes (however no distinction is made between balanced and unbalanced translocations)	No	Yes
Outputs average number of each type of rearrangement present per solution	No	No	Yes
Outputs all solutions	No	No	Yes

*The UniMoG bioinformatics tool offers a wide variety of models, for the purpose of this comparison, the rDCJ model is superior due to its increased level of biological accuracy (see Section 2.2.2.).

**The rearrangement scenario in the graphic consists of 13 rearrangements because no transpositions occur. Depending on the number of transpositions present in the solution (which can range from zero to four) the number of rearrangements per solution will range from nine to 13. No estimate is, however, given for the average number of transpositions per solution, so an accurate estimation for the number of rearrangements is not possible.

4.3.3.1. Genomic sorting model used by the tool and the types of rearrangements used by the model

The GRIMM bioinformatics tool makes use of the HP model [38]. This model is restricted to performing inversions (referred to as reversals), translocations (both balanced and unbalanced, however, no distinction between the two is made), and fissions and fusions. In addition to these rearrangements, the rDCJ model is able to execute transpositions and block-interchanges as well. UniMoG [60] offers a variety of models which can be selected, including the HP model, the DCJ model and the rDCJ model. For the purpose of this comparison, the results for the rDCJ model are

shown as this model is the most accurate from a biological perspective (see Section 2.2.2.). Genolve utilizes the rDCJ model, albeit an amended version of the model (see Sections 2.3.4. and 2.3.5. to revisit the modifications made to the model).

4.3.3.2. Number of rearrangements per solution

Genolve is able to achieve the transformation of MTZ13.12 into FJSA40.2 using four fewer rearrangements than the other two models. This is directly related to the number of transpositions used by Genolve.

Under the HP-model, a transposition can be replaced by three inversions or two translocations depending on whether the transpositions are inter- or intra- chromosomal.

The effect of a transposition operation confined to a single chromosome (i.e. sequence blocks are transposed from one location to a different location on the *same chromosome*) can be achieved using three inversion events. The output generated by GRIMM for the input genomes: target genome: [[1, 2, 3, 4, 5]] and source genome: [[1, 2, 4, 3, 5]], where sequence block [4] was transposed to in between sequence blocks [2] and [3], is shown in Fig 47.

Alternatively, the effects of an inter-chromosomal transposition (sequence block are transposed from one chromosome to another) are achievable using two translocation events. The output generated by GRIMM for the genomes, target genome: [[1, 2, 3], [4, 5, 6]] and source genome: [[1, 3], [4, 5, 2, 6]], where sequence block [2] is transposed to in between sequence blocks [5] and [6] are shown in Fig 48.

Similarly, under the rDCJ model, implemented in UniMoG inter-chromosomal transpositions (consisting of two DCJ operations) can be replaced by two balanced translocations (also consisting of two DCJ operations). In either case, the total number of DCJ operations used will remain constant and by extension, both solutions will be equally parsimonious.

5 genes Reversal Distance: 3

One optimal reversal scenario

Step	Description	
0	(Source)	1 2 4 3 5
1	Reversal	1 2 -4 3 5
2	Reversal	1 2 -4 -3 5
3	Reversal (Destination)	1 2 3 4 5

Fig 47. Output of the GRIMM bioinformatics tool for the input genome pair, target genome: $[[1, 2, 3, 4, 5]]$ and source genome: $[[1, 2, 4, 3, 5]]$. The sequence block [4] was transposed to in between sequence blocks [2] and [3]. The GRIMM tool, unable to perform transpositions, transforms the source genome into the target genome using three reversals or inversions. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.

2 chromosomes, 6 genes, 4 caps Multichromosomal Distance: 2

One optimal rearrangement scenario

Step	Description	
0	(Source)	-3 -1 4 5 2 6
1	Translocation	-3 -5 -4 1 2 6
2	Translocation (Destination)	-3 -2 -1 4 5 6

Fig 48. Output of the GRIMM bioinformatics tool for the input genome pair, target genome: $[[1, 2, 3], [4, 5, 6]]$ and source genome: $[[1, 3], [4, 5, 2, 6]]$. The sequence block [2] was transposed to inbetween sequence blocks [5] and [6]. The GRIMM tool, unable to perform transpositions, transforms the source genome into the target genome using two translocation events. Sequence blocks that will be affected by a rearrangement event are highlighted in yellow.

The output generated by UniMoG under the rDCJ model for the genomes, target genome: $[[1, 2, 3], [4, 5, 6]]$ and source genome: $[[1, 3], [4, 5, 2, 6]]$, where sequence block [2] is transposed to in between sequence blocks [5] and [6] are shown in Fig 49. Two balanced translocations are used to achieve the same effect as the transposition. First, sequence block [3] at the end of the first chromosome is exchanged with sequence blocks [2, 6] at the end of the second chromosome (first balanced translocation). Then sequence block [6] at the end of the first chromosome is exchanged with sequence block [3] at the end of the second chromosome (second balanced translocation).

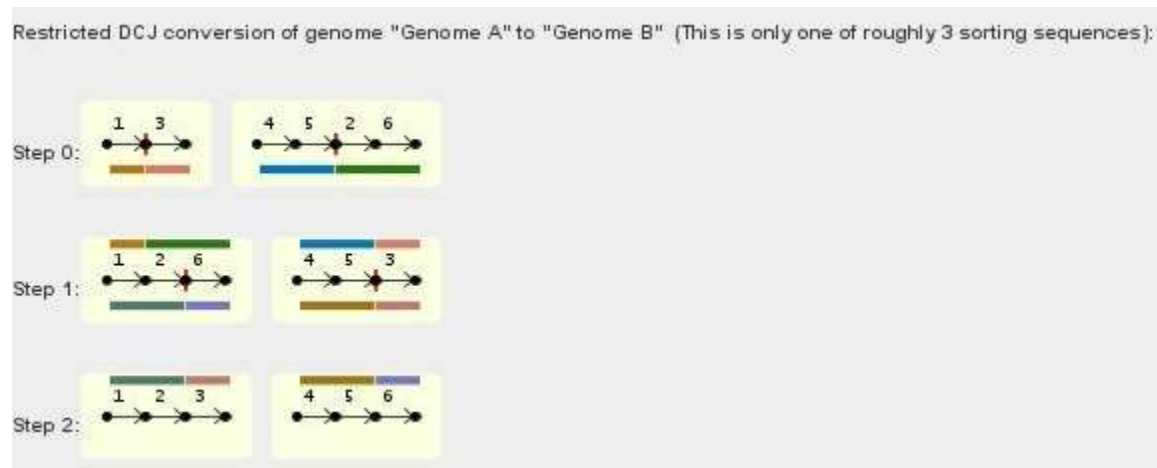


Fig 49. Sorting scenario generated by UniMoG, under the restricted DCJ model, for the input genome pair, target genome: $[[1, 2, 3], [4, 5, 6]]$ and source genome: $[[1, 3], [4, 5, 2, 6]]$. The sequence block [2] was transposed to in between sequence blocks [5] and [6]. The colouring under sequence blocks in one step corresponds to the colouring on top of the sequence blocks in the following step allowing easy identification of the location to which sequence blocks are relocated. The vertical red lines between genes indicate which adjacencies will be destroyed in the proceeding step.

If we refer to Fig 45, only one inversion (reversal) is applied in step one, the other 12 steps consist of translocations, indicating that all of the transpositions that occurred were inter-chromosomal and thus replaceable by two translocations. Genolve used four transpositions during the transformations process, if these four transpositions were each replaced by two translocations, the number of rearrangements per solution used by Genolve would also total 13. Thus, the difference between the number of rearrangements used by GRIMM and Genolve results from *GRIMM's inability to employ transpositions*.

Under the rDCJ model in UniMoG, in contrast to the HP-model used by GRIMM, transpositions can be executed. Therefore, the question arises why the tool generated a sorting scenario consisting of 13 rearrangements when a different sorting scenario (that the tool is capable of computing) would require fewer *rearrangements*. The answer to this question has likely to do with the amendment made to the rDCJ model implemented in Genolve.

In the original rDCJ model, all DCJ operations executed during the transformation are weighted equally. A transposition that consists of two DCJ operations thus has the same cost as two translocations. All three tools are designed to find the most parsimonious (lowest cost) solutions.

UniMoG, under the original rDCJ model, will thus identify a rearrangement scenario consisting of four transpositions or a rearrangement scenario consisting of eight translocations as equally parsimonious because both scenarios required the execution of the *same number of DCJ operations* (eight).

In contrast, in the amended rDCJ model implemented in Genolve, the two DCJ operations constituting a transposition are each assigned a cost of 0.5 (so as to ensure that transpositions operations do *not* have a higher cost than any of the other types of rearrangements, ensuring that the cost of one transposition is equivalent to the cost of one translocation). The implication of this is that Genolve will never identify the rearrangement scenario generated by UniMoG (Fig 46) as a most parsimonious solution because the total *cost* of four transpositions is half of that of eight translocations.

4.3.3.3. *Estimated number of most parsimonious solutions for the given input genome pair*

Another, very important, consequence of the amended version of the rDCJ model implemented in Genolve is a decreased solution space *in the presence of transpositions*.

As has been stated the solutions identified by Genolve for the transformation of *strain MTZ13.12 into strain FJSA40.2* will all include four transpositions (as well as three translocations a fusion event and a fission event). In contrast, the solutions identified by UniMoG under the rDCJ model will include the solution set identified by Genolve, as well as all the different permutations of rearrangement scenarios containing (i) zero transpositions and eight translocations, (ii) a single transposition and six translocations, (iii) two transpositions and four translocations, and (iv) three transpositions and two translocations, because all these solutions will consist of the same *number of DCJ operations*, namely eight.

GRIMM gives no estimate of the number of solutions. UniMoG, however, generated an estimate of 380,067,187,500 different rearrangement scenarios. This estimate is notably higher than the 161280 sorting scenarios found by Genolve.

This is, however, just a rough estimate as the as UniMoG outputs the same estimate of 380,067,197,500 rearrangement scenarios under the rDCJ model, the HP model and the DCJ model (see Fig 50). In reality, one would expect the number of possible scenarios under the DCJ model

to be higher than under the rDCJ model and the number of possible scenarios under the HP model to be the lower than under either the DCJ and rDCJ models as I will illustrate shortly. It is interesting to note that in Fig 50 a and c, where the results are shown for the rDCJ and HP models, respectively, the text on the image indicates that it is ‘rough’ - ‘This is one of *roughly* 380,067,197,500 sorting scenarios’. In contrast, under the DCJ model of which the results are depicted in Fig 50b, the text on the graphic states that ‘This is one of *at least* 380,067,197,500 sorting scenarios’.

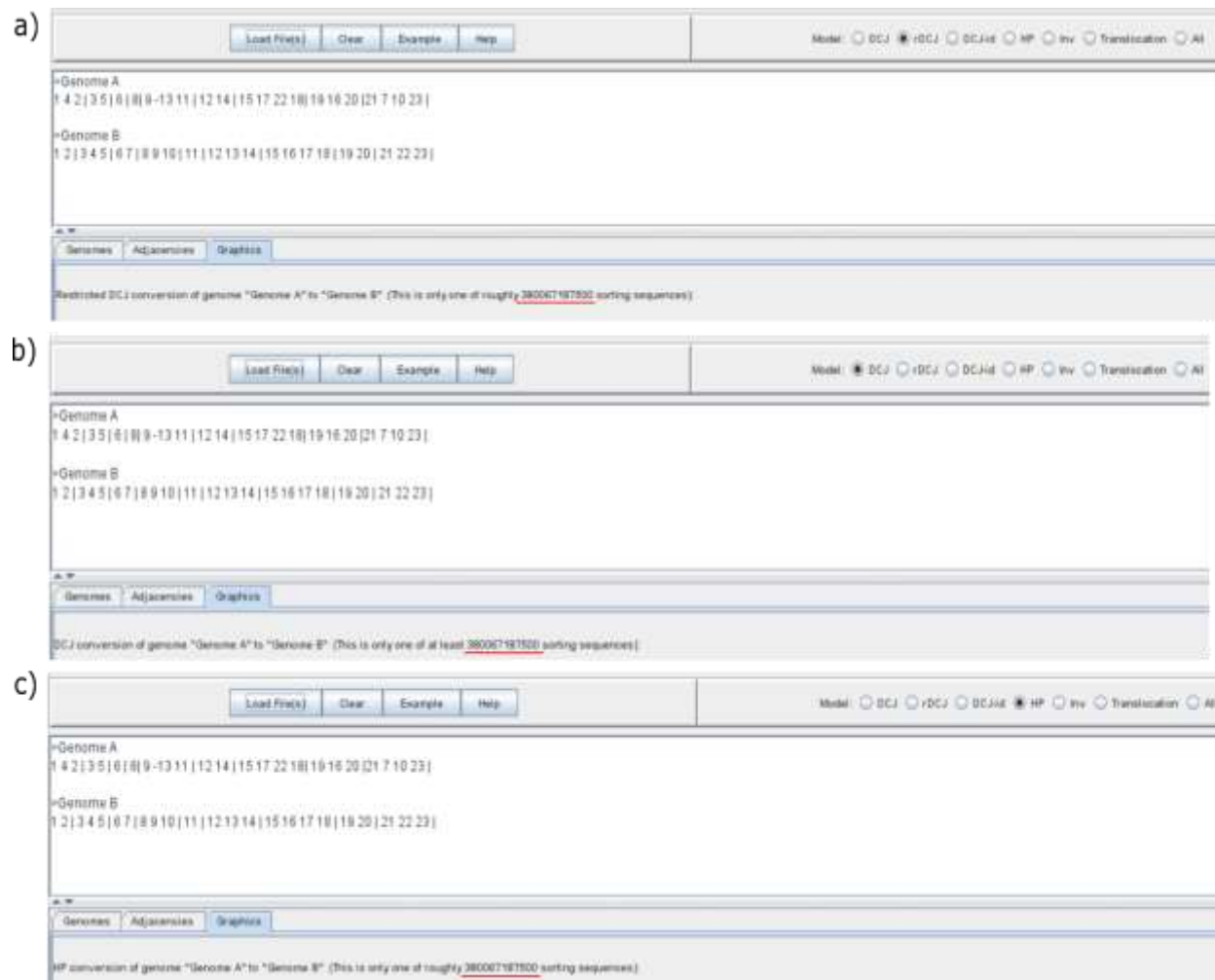


Fig 50. Screenshots of the results generated by UniMoG for the input genome in Fig 44 under the a) rDCJ, b) DCJ and c) HP model. Note that the screenshots exclude the graphics of the rearrangement scenario generated under each model as the purpose of this figure is to illustrate that under all three models, UniMoG states the rearrangement scenario shown is one of roughly 380067187500 (underlined in red) sorting scenarios.

To understand why, in the presence of transpositions, the HP model (which does not permit the execution of transpositions) is expected to generate fewer sorting scenarios than the rDCJ and DCJ models, and why the rDCJ model (which requires that circular chromosomes be reincorporated immediately after their formation) is expected to generate fewer sorting scenarios than the DCJ model, it is necessary to consider the following example:

Let us assume that transforming genome A into genome B can be accomplished by applying alternatively:

- (i) *Rearrangement set one* consisting of two inter-chromosomal transpositions (referred to as transpositions a and b), three inversions, one fusion and one fission or
- (ii) *Rearrangement set two* consisting of 4 translocations, three inversions, one fusion and one fission, or
- (iii) *Rearrangement set three* consisting of transposition a, two translocations, three inversions, one fusion and one fission, or
- (iv) *Rearrangement set four* consisting of transposition b, two translocations, three inversions, one fusion and one fission.

If it is assumed that all rearrangements are independent and as such, can occur in any order.

Because both transposition a and b are inter-chromosomal the effect of each individual transposition can be achieved by using two translocations.

Note that in each of the four rearrangement sets, the total number of DCJ operations that will be used to achieve the transformation is equal to nine.

Under the HP model, the solutions generated will consist of all the permutations of those rearrangement sets that *do not contain transpositions* because, under this model, transpositions are not permitted. Only rearrangement set two qualifies. There are nine *independent* rearrangements in this rearrangement set, which means there are 9! different ways in which the rearrangements can be ordered. Consequently, for this rearrangement set, under the HP-model, there will be 9! different solutions.

Under the rDCJ model, transpositions (consisting of two DCJ operations) are allowed, and all DCJ operations are *weighted equally*. Permutations of all four rearrangement sets will thus be present

in the solutions set as they will all consist of the same number of DCJ operations and as such, be identified as being *equally parsimonious*. However, due to the model's requirements that the two DCJ operations constituting a transposition occur in immediate succession, the number of ways the DCJ operations can be ordered will be affected by the number of transpositions present in the rearrangement set. In rearrangement set two, where no transpositions are present, the number of ways in which the nine DCJ operations can be ordered will be $9!$ (as was the case under the HP model). In rearrangement set one, however, there are two transpositions which means there will be only $7!$ different ways in which the DCJ operations can be arranged because there are two pairs of DCJ operations (the two transpositions) of which the order is fixed. For rearrangement sets three and four, containing one transposition each, the number of different permutations of the DCJ operations will equal $8!$ because there will be only one pair of DCJ operations of which the order is fixed.

The DCJ model places no restriction on when circular intermediates need to be reincorporated. For each of the four rearrangement sets, the number of ways in which the DCJ operations can be arranged per set (and as such the number of solutions) is $9!$.

As an interesting aside, the number of solutions that would exist under the amended rDCJ model implemented in Genolve was also considered. As has been stated, the amended rDCJ model recognizes one transposition and one translocation as have an *equal cost* despite the fact that transpositions consist of two DCJ operations and translocations of only one. In the example scenario considered above, the total cost of the rearrangements in the four rearrangement sets will be:

- (i) *Cost of rearrangement set one* = 7
- (ii) *Cost of rearrangement set two* = 9
- (iii) *Cost of rearrangement set three* = 8
- (iv) *Cost of rearrangement set four* = 8

Only the permutations of rearrangement set one would be identified as most parsimonious solutions. There will thus be $7!$ different solutions.

The table below summaries the results of this example

Table 17. The number of solutions generated from each of the rearrangement sets and the total number of solutions generated under the HP- rDCJ-, DCJ- and amended rDCJ- model.

Rearrangement sets	Number of solutions under the HP model	Number of solutions under the rDCJ model	Number of solutions under the DCJ model	Number of solutions under the amended rDCJ model
One	None	7!	9!	7!
Two	9!	9!	9!	None
Three	None	8!	9!	None
Four	None	8!	9!	None
Total	362880	448560	1451520	5040

This example is very simplistic in that

1. all rearrangements are independent and
2. no set of rearrangements can be replaced by another set of rearrangements of the same size that achieves the same result.

Any factor that makes the example more complicated will, however, have *the same effect on all of the models* – the presence of dependent rearrangements (*i.e.*, where it is required that one rearrangement always proceed another specific rearrangement) will decrease the number of solutions over all the models. Alternatively, if one set of rearrangements is replicable by another set of rearrangements, this will increase the number of solutions generated under each model.

The example clearly shows that the number of solutions generated under the HP-model is *less* than under the rDCJ model and the number generated under the rDCJ model is less than under the DCJ model.

The ‘rough estimate’ of the number of sorting scenarios that UniMoG is thus a *very rough* estimate. It is likely that under the rDCJ model, the tool, if it were capable of computing all possible most parsimonious solutions, would identify considerably fewer solutions than the reported 380067187500 estimate.

In either event, the fact remains that the amended rDCJ model implemented in Genolve not only increases the biological accuracy of the model, but has the added benefit of decreasing the solution space when transpositions are present.

4.3.3.4. Identification of rearrangements

Unlike UniMoG, the output of which consists of only DCJ operations, GRIMM and Genolve both identifies the type of rearrangement event each sorting operation represents. Genolve distinguishes between inversions (inv), transpositions (trp0, trp1), balanced translocations (b_trl), unbalanced translocations (u_trl), fissions (fis), fusions (fus) and block-interchanges (trp0, trp2). GRIMM distinguishes between reversals, translocations, fissions and fusions. Transpositions and block-interchanges are not permitted under the HP-model implemented by the tool, and as such does not require identification. GRIMM does not make any distinction between balanced and unbalanced translocations.

4.3.3.5. Outputs average number of each type of rearrangement present per solution

Only Genolve gives the average occurrences of each type of rearrangement per solution. As was seen above, these metrics allow valuable information to be extracted from the solution set, allowing the deduction of the number and types of rearrangements present in different portions of the solution set.

4.3.3.6. Outputs all most parsimonious solutions

GRIMM and UniMoG generate only one of a large number of most parsimonious rearrangement scenarios. The output of Genolve, in contrast, includes all most parsimonious solutions identified by the tool.

4.3.4. Analysis of sequence regions at the breakpoints identified by Mauve

In this chapter, Genolve was used to look into the large-scale evolutionary history of two *S. cerevisiae* yeast strains. In addition to the reorganization of the genome during the evolutionary process, it is possible that some of the genes or other genomic features, present in the more ancient FJSA40.2 strain, may have been disrupted in the evolved MTZ13.12 strain. This disruption would occur if a breakpoint (start position or end position of a sequence block) in the FJSA40.2 strain's

genome occurred within a gene region or other genomic feature, so that some portion is relocated to elsewhere in the genome.

Determining whether the evolutionary process did result in the disruption of any genetic elements, and if so, analysing such instances would be of interest, yielding further insight into the evolutionary history between FJSA40.2 and MTZ13.12.

The genome assemblies of FJSA40.2 and MTZ13.12 are unannotated. As such, it is impossible to identify the type of sequence regions in which breakpoints occur, directly from the assemblies themselves. The identification of the region surrounding the breakpoints (the breakpoint region), is, however, possible by using the BLAST tool [87]. The breakpoint region can be aligned to the well-annotated *S. cerevisiae* reference genome. If it maps to a coding domain sequence or some other genetic element, it is indicative of the disruption of this region during the evolution of FJSA40.2 into MTZ13.12.

CAF4 is an example of a gene that was disrupted due to the large-scale genome rearrangements that occurred during the evolution of FJSA40.2 into MTZ13.12. The gene is located on chromosome 11 of the FJSA40.2 genome at position 512073bp to 514010bp. The gene occurs in the reverse orientation on the chromosome so that the end of the gene precedes the start. The breakpoint, identified by Mauve [62], resulted in gene disruption at position 512381bp, 308bp away from the end of the gene.

Fig 51 shows a graphic representation of the gene within the *S. cerevisiae* reference genome, with a red marker indicating the location of the breakpoint.



Fig 51. Position of the CAF4 gene within chromosome 11 of the *Saccharomyces cerevisiae* reference genome. A red marker shows the location of the breakpoint within the gene. (Figure obtained from the NCBI gene graphic viewer platform).

When aligned to chromosome 11 of the MTZ13.12 genome, only the latter part of the gene maps to the chromosome at position 524777bp to 525085bp. This graphic illustration generated by the BLAST tool [87] of the partial mapping is shown in Fig 52.

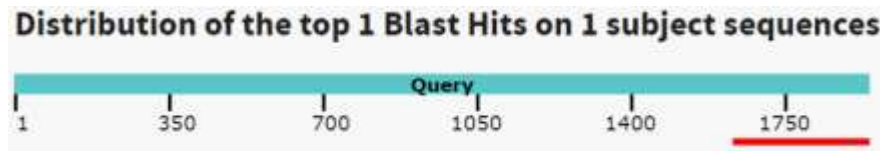


Fig 52. Graphic summary generated by the BLAST tool for the alignment of the CAF4 gene (the query sequence shown in blue) to chromosome 11 of the MTZ13.12 Saccharomyces cerevisiae strain (shown in red). The latter portion of the gene maps to position 524777bp – 525085bp on the MTZ13.12 chromosome.

The remaining, larger, portion of the gene was found on chromosome 10 of the MTZ13.12 genome (Fig 53) at position 280522bp to 282218bp. During the process of genome rearrangement, this portion of the gene thus relocated from its initial position on chromosome 11 in the FJSA40.2 genome to a new position on chromosome 10 in the evolved MTZ13.12 genome.



Fig 53. Graphic summary generated by the BLAST tool for the alignment of the CAF4 gene (the query sequence shown in blue) to chromosome 10 of the MTZ13.12 Saccharomyces cerevisiae strain (shown in red). The front portion of the gene maps to position 280522bp – 282218bp on the MTZ13.12 chromosome.

CAF4 is a non-essential gene that codes for a protein which plays a role in mitochondrial fission [88]. Interestingly, the gene has a paralog, MDV1, that arose from whole-genome duplication [88]. Aligning the MDV1 gene to the MTZ13.12 genome revealed that the gene was still intact and present on chromosome 10 at position 192475bp to 194619bp.

The relationship between CAF4 and MDV1 may suggest that, even when CAF4 is absent, MDV1 can compensate for any lost functionality. The protein sequence of the latter was aligned to the former using the EMBOSS Needle tool [89] (which employs the Needleman-Wunsch algorithm for the local alignment of sequences) of the EMBL-EBI, the sequence identity is only 33.3%, the sequence similarity 50.4% and the alignment score, 1029.5. On a protein sequence level, divergence between MDV1 and CAF4 has clearly occurred since the whole genome duplication event. Indeed, despite the fact that both genes function as adaptor proteins during the mitochondrial fission process, functional and phylogenetic analyses have indicated that the mitochondrial activity of CAF4 has diverged from that of MDV1 [88].

The CAF4 gene is non-essential and the loss of the gene from the MTZ13.12 genome will therefore not have a negative impact on the immediate survival of the strain. However, it is possible that the loss of fitness as a result of the disruption of the CAF4 gene was severe enough to contribute to natural selection against the MTZ13.12 strain, explaining why the *S. cerevisiae* reference genome does possess the gene.

Further investigation included looking at the component of the CAF4 gene containing the start codon (the portion of the gene relocated to chromosome 10) in the MTZ13.12 genome. As the start codon remained intact, it is possible that the gene portion may have fused with the latter part of some other gene portion, resulting in a novel gene with unique functionality. The gene region on chromosome 10 was analysed to find the first *in frame* stop codon proceeding the gene component. This whole region, start to stop codon, was then BLASTed against the entire NCBI database to determine whether the gene mapped to a characterized gene in any other species. All results contained only partial mappings (with a maximum coverage of 88%) of the CAF4 gene component of the ‘new’ gene to the CAF4 gene in numerous other genomes. It can thus be assumed that the relocation of the CAF4 gene component containing the start codon did not result in new gene functionality.

Chapter 5: Conclusion and Future prospects

5.1. Conclusion

The project aimed to develop a bioinformatics tool capable of identifying all the different (biologically feasible) ways in which one genome may have evolved into another. We chose to focus on structural changes, not on indels or SNPs, for which many tools exist. The tool that was developed, Genolve, achieved this identification of structural changes.

Genolve employs a recursive algorithm that utilizes an amended version of the DCJ model (allowing the assignment of the correct cost to transposition and block-interchange operations) to identify the complete set of most parsimonious rearrangement scenarios that can account for the evolution of one genome into another. The algorithm successfully transformed a source genome into its associated target genome 100% of the time in all tested simulations (which numbered over 4.8 million input genome pairs).

The percentage of times the ‘true’ evolutionary scenario (the series of rearrangements used by the Evolver to create the source genome) was present in the solution set generated by Genolve was investigated. Genolve was often unable to identify the ‘true’ evolutionary scenario as a possible solution. The most likely cause for this is the inherent nature of the DCJ model, where a DCJ operation has to create *at least* one final-state adjacency. The higher the number of rearrangements that influences the same sequence blocks during the evolutionary process (or the creation of the source genome from the target genome by the Evolver), the less likely it becomes that Genolve will be able to identify the true series of rearrangements that affect those sequence blocks. This points to a limitation of Genolve – that its accuracy in resolving the events that transpired, deteriorates in rearrangement hotspots.

An adjustable weighting system, was incorporated into the Genolve algorithm, allowing users to input the expected frequency of occurrence of each of the different type of rearrangements for a specific input genome pair. It was expected that if the given frequencies of occurrence were correct, it increases the likelihood that the ‘true’ evolutionary scenario was present in the solution set generated by Genolve. This was however not the case, with the weighting system that assigns an equal weight to all the types of rearrangements outperforming the weighing system, which

incorporates the correct expected frequencies of occurrence. The calculation with which the expected frequencies of occurrence for the rearrangement types is transformed into edge weights for the network of solutions is relatively simple. Reinvestigation of the relationship between expected frequencies of occurrence and edge weights (perhaps by setting up a series of Monte-Carlo simulations) may result in a more complex and accurate calculation for transforming the former into the later.

Alternatively, we can accept that the one-to-one ratio is the superior option and not allow for the adjustment of edge weights. This would open an avenue for redesign and adjustment of the algorithm, which has the potential to be extremely advantageous from both a runtime and memory usage perspective. Under a one-to-one weighting ratio, the most parsimonious, the "cheapest" path will *always* be the *shortest* path. This is not the case under the same-as-solution weighting ratio. Allowing the incorporation of user-adjustable expected frequencies necessitates the calculation (and storage) of *all* paths to get from the source genome to the target genome since, depending on the weights assigned to the different types of operations, some paths may be cheaper than others despite being longer (consisting of more operations). If only the one-to-one weighting ratio is used, cheaper paths are *always* the shorter paths; thus, it is unnecessary to continue along a path that has exceeded this minimum path length. Paths that exceed the minimum length can thus be aborted before the transformation process into the target genome is complete, improving execution efficiency.

Another point of investigation was the number of solutions per solution set, which was expected to increase with increasing levels of genomic complexity. This was found to be the case. There were, however, high levels of variation *within* levels of genomic complexity. The solutions within some solutions sets consist of more rearrangements (i.e. more rearrangements separate source from target genome). In cases where a higher number of rearrangements coincides with a larger percentage of independent rearrangements, there will be a higher number of different ways in which the rearrangements can be ordered (more permutations of the rearrangements) and consequently more solutions per solution set.

The average number of rearrangements per solution set was also expected to increase with genomic complexity, and this was also found to be the case. A large amount of variation once again existed between the average number of rearrangements *within* levels of genomic complexity. As

mentioned above, this has an impact on the number of solutions within a solution set. There are two factors contributing to the variation in the number of operations that were observed. The first is the effects of the Evolver that was used to generate the input genomes. As was illustrated in Fig 33, there may be cases where the Evolver reverses a previously applied rearrangement, thereby decreasing the number of rearrangements required for the transformation process. The second factor has to do with the DCJ operations applied to the source and intermediate genomes during transformation into the target genome. If a large number of DCJ operations result in *two* final-state adjacencies, less DCJ operations will be required for successful transformation into the target genome. In contrast, if each DCJ operation creates only *one* final-state adjacency, as many DCJ operations as there are non-final state adjacencies in the source genome would be required to transform it into the target genome. The average number of operations per solution within a solution set is thus dependent on the number of DCJ operations that create only one final-state adjacency.

5.1.1. Computational and memory complexity of Genolve

The final two factors that were investigated were the runtime and memory usage of Genolve. Both the time and space complexity of the algorithm was shown to be $O(n!)$. As such, both the runtime and memory usage of Genolve increased with genomic complexity. A large amount of variation was, however, observable within levels of genomic complexity. Both runtime and memory usage of Genolve are influenced by (i) the number of DCJ operations that creates only one final-state adjacency, (ii) the percentage of DCJ operations that are independent and (iii) the extent to which there exists a convergence of paths within the network of solutions, all three of which contribute to the variability within levels of genomic complexity. In addition to these three factors, the runtime of Genolve will also be influenced by the CPU's variability in processing speed.

Genolve's ability to perform on biological data was also evaluated by running the tool on a pair of related strains of *S. cerevisiae*. Genolve successfully identified the structural change path in the transformation of the modern strain into the primitive strain, identifying 161280 different rearrangement scenarios that could have resulted in the evolution of the latter into the former. The summary metrics included in the output of Genolve, namely the number of solutions within the solution set, the number of rearrangements per solution and the average number of each of the

different types of rearrangement per solution allowed the tool to be used to gain novel insights into the types of rearrangement scenarios that could describe this evolutionary process. It was deduced that (i) the rearrangements present in the various solutions are not all independent, some rearrangement is dependent on the prior occurrence of one or more other rearrangements, (ii) 11.67% of the solutions consists of four transpositions, three balanced translocations, one fusion and one fission, 44.17% of the solutions consists of four transpositions, two balanced translocations, one unbalanced translocation, one fusion and one fission, and 44.167% of the solutions consists of four transpositions, one balanced translocation, two unbalanced translocations, one fusion and one fission.

The modified version of the rDCJ model implemented in Genolve, in addition to increasing the biological accuracy of the model, also results in a smaller solution set than would be generated under the original rDCJ model.

Neither GRIMM nor UniMoG outputs the entire set of most parsimonious rearrangement scenarios as Genolve does, nor do they output the average number of each type of rearrangement present per solution. Unlike Genolve and GRIMM, UniMoG shows only the DCJ operations used for genomic sorting, not identifying the types of rearrangement events represented by these operations, making analyses of the scenario more tedious. Additionally, the rearrangement model options offered by UniMoG, as well as the HP-model implemented in GRIMM, are not as accurate, from a biological perspective, as the modified rDCJ model implemented in Genolve.

5.2 Future prospects

Two areas for future improvement of Genolve has been alluded to in the above section. The first concerns the weighting system of Genolve. The opportunity exists to devise a more accurate weighting system based on the expected frequency of occurrence of the different types of rearrangements. Additionally, when the expected frequency of occurrence of all the types of rearrangements are equivalent (*i.e.*, a one-to-one weighting ratio exists), the algorithm can be redesigned to terminate the calculation of any path that exceeds the minimum length of paths encountered at that point, resulting in a more time and memory-efficient algorithm.

The second area for improvement is Genolve's ability to resolve rearrangement hotspots accurately. As was explained, Genolve is limited by the inherent nature of the DCJ model. Further research into possible amendments to the DCJ model that will extend its utility or alternatively, the development of a new model for genome rearrangement that does not have this limitation, will improve the accuracy of Genolve.

In addition to the improvements mentioned above, there are two additional factors that, if incorporated into the Genolve algorithm, will increase its utility significantly. The first is incorporating a strategy that allows Genolve to account not only for genomic rearrangement events, but also for genomic content-altering events (namely insertions, deletions and duplications). The second is the decrease in the solution space of the output of Genolve by incorporating biological constraints.

5.2.1 Genome content altering events

Genolve is able to account for all of the types of large-scale events that affect the order in which sequence regions occur in a genome, *i.e.*, the rearrangement events. It does not, however, account for the large-scale content altering events, namely insertions, deletions and duplications. Incorporating methods with which to account for these types of events into Genolve would increase the utility of the tool. Various methods to account for one or a combination of content-altering types of events have been devised.

5.2.1.1. Duplications

Sankoff [90] addressed the problem of duplications by introducing the *exemplar* method. Briefly, this method relies on the selection of one copy of any given gene in both genomes (removing the rest) so that the selected copies minimize the number of rearrangements required to sort the resulting *exemplar permutations* (Fig 54a). Two related strategies followed the *exemplar* method, namely the *intermediate strategy* [91] and the *maximum matching* strategy [92]. The former constitutes selecting the same number of copies of a given gene in the respective genomes, each copy in the one being matched to a copy in the other (allowing sorting algorithms to treat them as distinct gene pairs) (Fig 54b). These matches are selected to minimize the number of

rearrangements required to solve the genomic sorting problem. The *maximum matching strategy* functions in a similar fashion, differing in that the maximum number of gene copies are selected so that remaining copies of a gene in only one of the respective genomes need to be removed. As with the previous strategy, matches are made between copies in order to minimize the number of sorting operations required (Fig 54c). See [93] for a summary of the results obtained for each of the strategies using different sets of rearrangement operations.

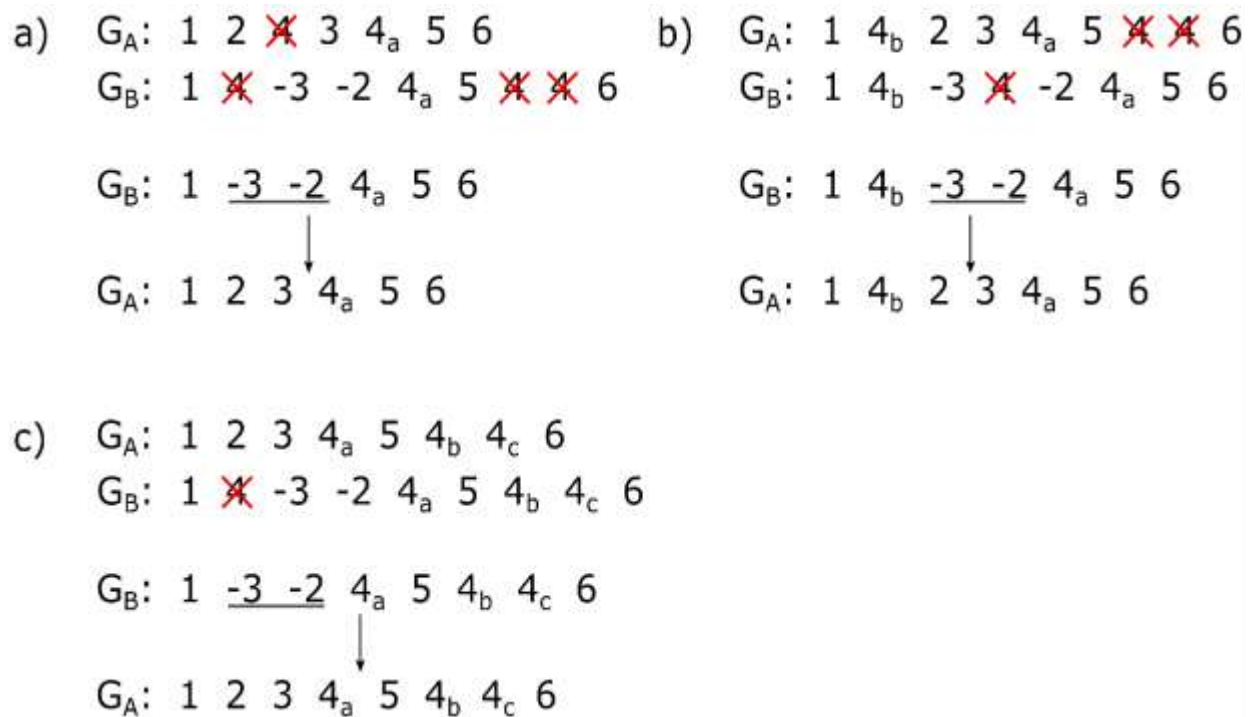


Fig 54. Illustration of three different approaches to dealing with duplications prior to genome sorting. In each example one or more copies of gene '4' in G_A is matched to one or more copies of the gene in G_B (with unmatched copies being removed), prior to the sorting of G_B into G_A . The removal of a gene copy is shown by a red cross over the gene. The letters in the subscripts of gene '4' show which copies in the two genomes have been matched. a) Example of the 'exemplar method', where one copy of each gene duplicate per genome is selected and matched to the duplicate in the other genome. The rest of the copies are removed. b) Example of the 'intermediate strategy' where a number of copies of a duplicative gene are selected in one genome and matches to copies in a second genome. The unmatched copies of the gene in both genomes are removed. c) Example of the 'maximum matching strategy', where the maximum number of duplicates of a gene are selected in one genome and matched to gene copies in a second genome, while the unmatched copies are removed.

5.2.1.2. Using ‘ghost’ vertices

An extension of the original DCJ model that permits insertions, deletions and duplications was introduced in [94]. The model utilizes what is termed ‘ghost’ vertices to account for missing genes. Three separate cases are considered, namely, (i) insertions and deletions, (ii) duplications with equal genic content, and (iii) duplications with unequal genic content. In the case of insertions and deletions, ghost vertices, representing the missing gene extremities are introduced into the adjacency graph, enabling genomic sorting via DCJ operations, to proceed as normal - the ghost vertices being treated as any other vertex (Fig 55). It is shown that a surcharge of one to the genomic distance, for every 1-cycle containing a ghost vertex, is required to ensure that the incorporation of ghost vertices do not disrupt the existing DCJ distance structure. The addition of ghost vertices is done to minimize the distance between the genomes. In Fig 55 the genes introduced into the genome as ghost vertices are 2 and 3. The way in which the ghost vertices are introduced in Fig 55a compared to Fig 55b allows for the sorting process of the former to occur in a fewer number of steps than that of the latter.

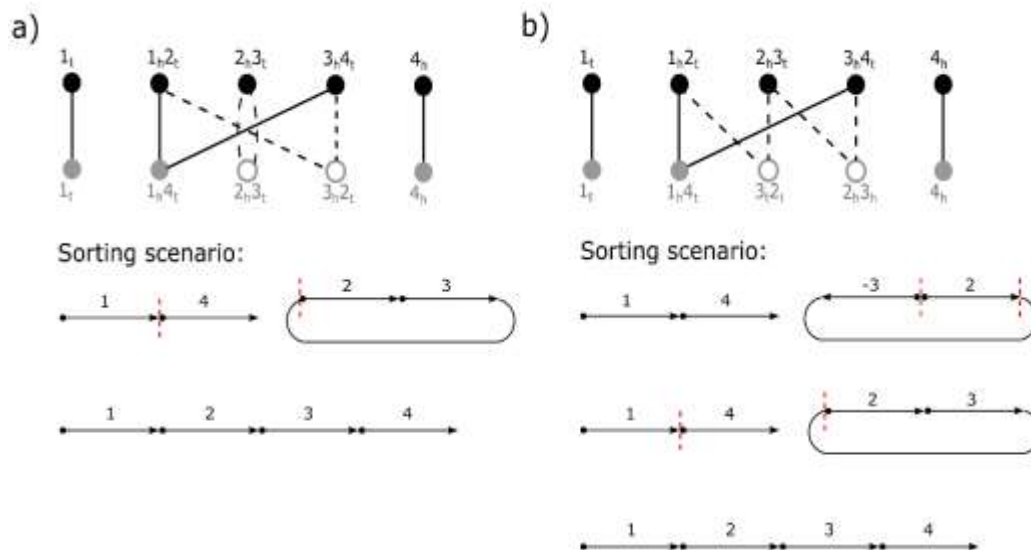


Fig 55. Adjacency graphs for the genomes of which the genic content is not equal, $G_A = \{(\circ, 1, 2, 3, 4, \circ)\}$ is represented by black vertices, and $G_B = \{(\circ, 1, 4, \circ)\}$ by grey vertices. ‘Ghost’ vertices are used to represent adjacencies between gene extremities not present in G_B . **a)** ghost vertices are used to form adjacencies $\{2_h 3_t\}$ and $\{3_h 2_t\}$. This results in an adjacency graph consisting of two paths of length one, a cycle and a one-cycle containing a ghost vertex. **b)** ghost vertices are used to form different adjacencies, $\{3_t 2_t\}$ and $\{2_h 3_h\}$, thereby avoiding the formation of a one-cycle containing a ghost vertex.

Concerning the presence of duplicative genes for which the number of duplicates per genome are the same, those in the original genome are simply matched to copies in the target genome in such a way as to minimize the number of operations required to sort the one genome into the other. In cases where one genome contains more copies of a gene than the other, ghost vertices are introduced as in the case of insertions and deletions.

5.2.1.3. Calculating the intermediate

A different approach to sorting genomes by DCJ and indel operations was introduced by [95]. Their algorithm differs from the above in that (i) it is unable to account for genomes containing duplicated genes and (ii) it does not rely on the use of ‘ghost’ vertices to account for insertion and deletion events. Instead, they transform (using DCJ and deletion operations) both genomes G_A and G_B into an intermediate genome G_I that consists of only those genes present in both genomes. This results in an operation set s_1 which consists of the sequence of operations applied to G_B (Fig 56a) and an operation set s_2 which is the sequence of operations applied to G_A (Fig 56b), resulting in both genomes being sorted into G_I .

Finding an optimal sorting sequence to sort G_B into G_A then simply requires sorting G_B into the intermediary genome G_I and back-tracing the operations used to sort G_A into G_I (Fig 56c). The optimal sorting sequence is thus given by $s_1 s_2^{-1}$ and is calculable in linear time [95].

They also show that there are two different ways to approach the sorting problem, or two different *types* of sorting scenarios. The first is minimizing the number of DCJ operations relative to the number of indels utilized, permitting more indel operations if it means using less DCJ operations. The second is minimizing the number of indel operations relative to the number of DCJ operations used, ensuring as few as possible indels are performed.

5.2.1.4. Restricted DCJ-indel sorting for linear genomes

A similar model was introduced by [96] with an additional restriction that the formation of a circular chromosome via a DCJ operation has to be preceded directly by its reincorporation, i.e.

no other DCJ operation or indel operation is permitted to occur between the circularization and reincorporation steps. This use of restricted DCJ operations increases the biological applicability of this approach regarding linear genomes. Only an upper bound for the restricted DCJ-indel distance was given, posing the open question as to whether the reduction of this bound is possible, so that the general and restricted DCJ-indel distances can be shown to be equal.

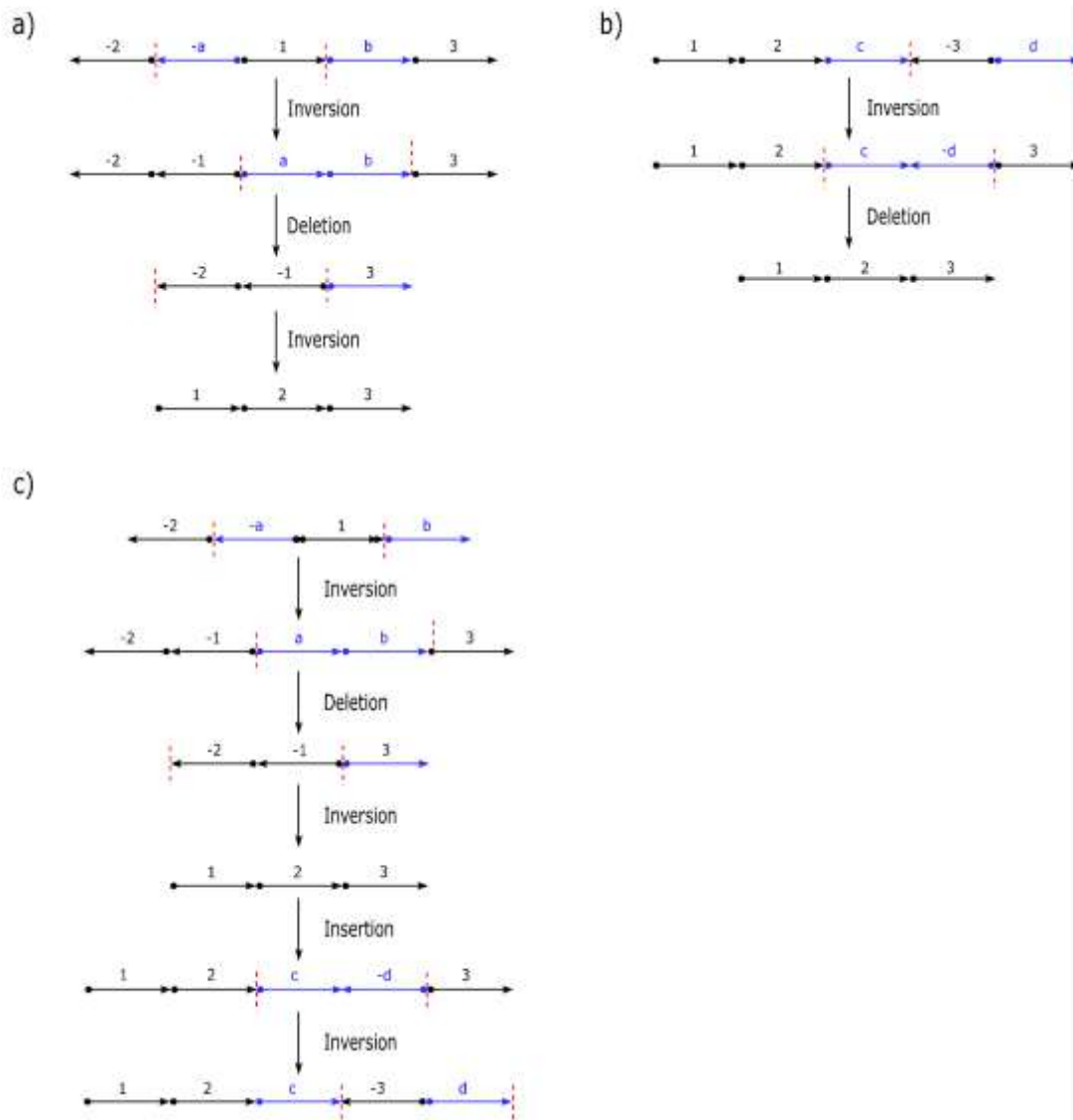


Fig 56. Adjacency graphs for the genomes of which the genic content is not equal, $G_A = \{\{\circ, 1, 2, 3, 4, \circ\}\}$ is represented by black vertices, and $G_B = \{\{\circ, 1, 4, \circ\}\}$ by grey vertices. 'Ghost' vertices are used to represent adjacencies between gene extremities not present in G_B . a) ghost vertices are used to form adjacencies $\{2h3t\}$ and $\{3h2t\}$. This results in an adjacency graph consisting of two paths of length one, a cycle and a one-cycle containing a ghost vertex. b) ghost vertices are used to form different adjacencies, $\{3t2t\}$ and $\{2h3h\}$, thereby avoiding the formation of a one-cycle containing a ghost vertex.

Further exploration of the restricted DCJ-indel model [97] yielded results showing that the genomic distance under both the restricted and the general DCJ-indel sorting model are the same, both being computable in linear time. Additionally, a simpler DCJ-indel sorting algorithm with a runtime of $O(n \log(n))$ was introduced.

5.2.1.5. Treating deletions as DCJ operations

A different method of sorting by DCJ operations and indels [98,99] treats the deletion of an interval of genes from a chromosome as a DCJ operation. Similar to the modeling of transpositions and block-interchanges, a region of DNA is excised and circularized in the first step, but instead of being linearized and reinserted elsewhere in the genome, the circular chromosome gets removed (deleted).

For example the following DCJ-deletion operation: $(\{\circ, 1_t\}, \{\underline{1_h}, \underline{2_t}\}, \{2_h, 3_t\}, \{\underline{3_h}, \underline{4_t}\}, \{4_h, \circ\}) \rightarrow (\{\circ, 1_t\}, \{1_h, 4_t\}, \{4_h, \circ\}, \{2_h, 3_t\}, \{3_h, 2_t\})$ produces a circular intermediate consisting of genes 2 and 3. Instead of the subsequent reincorporation of this intermediate, it gets ‘marked’ for removal. The deletion of all circular chromosomes marked for removal is performed in the final step of the transformation process. This removal of genes allows the creation/generation of G' from G . The insertion of an interval of genes to obtain G from G' is simply the inverse of the deletion operations corresponding to the generation of G' from G .

The transformation of a genome G_B into a genome G_A comprises three steps:

1. The insertion of chromosomes consisting of genes present in G_A but absent from G_B into G_B to yield the genome G_B' .
2. Sorting G_B' via normal DCJ operations to produce G_A' .
3. Deleting the genes from G_A' that are present in G_B but absent from G_A to produce G_A .

G' is defined as the *completion* of G , and correspondingly (G_B', G_A') is the *completion pair* of the genomes G_B and G_A . An *optimal* completion pair is one that enables the attainment of a minimum distance for sorting a genome pair (G_B, G_A) by DCJ and indels, and is denoted by (G_B^*, G_A^*) .

(G_B^*, G_A^*) can be constructed via the direct analysis of the breakpoint graph $BP(G_A, G_B)$. Once an optimal completion (G_B^*, G_A^*) has been obtained, the problem reverts to the simple version of transforming G_B^* into G_A^* using the linear time sorting algorithm introduced by Bergeron et al. [55] for sorting genomes by DCJ operations.

5.2.1.6. *Other approaches*

Various other attempts have been made to incorporate indels and or duplications into sorting algorithms. There is a heuristic algorithm for sorting unichromosomal genomes by reversals, block-interchanges, duplications and deletions [100], later extended to sort multi-chromosomal genomes [101]. An iterative algorithm based on a new trajectory graph model, allowing for the sorting of genomes with rearrangement operations and segmental duplications was introduced in [102]. This model is however based on the assumption that duplications events always proceed or precede all DCJ operations, which is not the case in every biological setting.

A different model allowing for SNPs in conjunction with DCJ operations, deletions and duplication was developed by [103]. The authors introduce a novel data structure, the *history graph* which contains partial order data on the sequence of events. The aim is the identification of a sequence of events consistent with the input history graph that would minimize the set of SNPs and DCJ operations required, each of which is associated with a non-zero cost. Deletions and whole genome duplications are given a cost of zero. A polynomial tractable bound for the cost was provided by the authors.

5.2.2 *Biological constraints*

Despite the success of Genolve in identifying all most parsimonious evolutionary scenarios between two genomes, the vastness of the number of possibilities that often exists describing this, cannot be discounted.

The question of feasibility in terms of in-depth analysis of the entirety of a very large solution set arises. Conducting such in depth analyses may require employing some strategy to decrease solution space.

The problem is that decreasing solution space requires the dismissal of individual solutions from a set. Unless the grounds for dismissal are biological in nature, the result will simply be an incomplete picture of what may have transpired between two evolutionary related genomes.

Fortunately, various methods of incorporating different types of biological data in the study of large-scale genomic evolution has been considered.

5.2.2.1. Inversion lengths and symmetric inversion

A study using eight *Yersinia* genomes to study the evolution of genome structure [104] resulted in a number of interesting findings; (i) the lengths of all inversions were shorter than would be expected under the neutral model, (ii) there was an overrepresentation of what was termed ‘symmetric inversions’, which are inversions of which the end points are equidistant from the origin of chromosomal replication and (iii) there was an increased likelihood for the occurrence of inversions of which the endpoints are proximal to the origin of replication. [105,106] showed that there are numerous cases in which short inversions are found to affect only a single gene, this contrasts with the null hypothesis that the two respective endpoints of an inversion event occur at random and independently [104].

The problem of sorting genomes using only symmetric or almost-symmetric inversion operations was introduced in [107]. An algorithm for sorting by weighted inversions, with weight assigned with respect to the length of the inversion and its asymmetry followed [108].

5.2.2.2. The length of intergenic regions

Another biological constraint that has been considered, pertains to the length of intergenic regions between genes. It was shown that, according to the Nadeau-Taylor model of uniform breakage [109], it is more likely that a breakpoint will occur in a longer intergenic region given the increased ‘fragility’ of these regions relative to coding regions [110]. “Fragility” was a term used by the authors, and it is not clear whether this refers to a locally decondensed state of chromatin, or reflected another physical property of the genome. From this, DCJ sorting algorithms able to incorporate the length of intergenic regions were developed [111,112]. These algorithms take as

input, not only the two genomes under comparison but also the intergenic *sizes* between the genes that constitute them. One genome is then transformed into the other using weighted DCJ (wDCJ) operations.

The genomes themselves are also weighted, where each edge of a genome G is assigned a weight corresponding to the length of the intergenic region it represents. $W(G)$ is then the sum of all the weights of the edges of the genome. A wDCJ operation acts in a similar fashion to a DCJ operation with the additional incorporation of the weight variable of the edges being acted upon.

Given a genome G , a DCJ operation deletes edges ab and cd and forms edges ac and bd or edges ad and bc . If the genome is *weighted* then the wDCJ operation has the additional requirement that $w(ac) + w(bd) = w(ab) + w(cd)$ or $w(ad) + w(bc) = w(ab) + w(cd)$. The algorithm in [94] is somewhat limited in its biological applicability in that it requires $W(G_A) = W(G_B)$, that is to say, the total size of the intergenic regions of the two respective genomes should be conserved. This is not the case in biological data where even closely related genomes have varied intergenic lengths. The algorithm presented in [112] does not have this restriction and is thus of more biological relevance. The authors present a polynomial time algorithm to calculate a wDCJ scenario with insertions and deletions permitted in intergenic regions.

5.2.2.3. *The physical proximity of breakpoints*

The importance of the physical proximity of breakpoints in the genomes was highlighted in a study making use of Hi-C data (data describing the 3D organization of chromatin) [113]. It was shown that loci positioned far from one another when considering their linear ordering on the human chromosome, but close together on the mouse chromosome, are in close physical proximity on the human chromosome when considering the 3D model of the chromosome [113].

In a different study of genome rearrangements [114] it was discovered that the physical locations of pairs of breakpoints involved in rearrangements are nearer one another than would be expected by chance. This was true for both inter- and intra- chromosomal rearrangements and was consistent

for multiple cell types across multiple laboratories [114]. Other studies noting the importance of the role that the 3D spatial proximity of breakpoints play in their formation include [115,116].

A DCJ sorting algorithm that incorporates a binary weight function that assigns a weight to each DCJ operation based on its likelihood of occurrence was outlined in [117]. The purpose of the weight function is to model positional constraints based on the assumption that operations involving adjacencies with a close proximity in the physical structure of the genome, occur in higher frequency.

Adjacencies in the genome are grouped into classes that have higher likelihoods of swapping endpoints due to their occupation of the same spatial region in the 3D conformation of the genome. A ‘color’ (labeled by an alphabetical letter) is assigned to each adjacency, based on their locality in the 3D structure of the genome (Fig 57).

As has been stated, the weight function is binary in nature, with an operation falling into either the category of likely or rare. If the colors of the adjacencies involved in the operation are the same, the operation falls into the ‘likely’ category and is given a weight of zero, else it is given a weight of 1. The total weight of a sorting scenarios is simply the sum of the weights of the operations that constitutes it.

The authors present a polynomial time algorithm ($O(n^4)$) to solve what they introduce as the minimum local parsimonious scenario for sorting genome A into genome B - the problem of finding the most parsimonious scenario with the lowest weight.

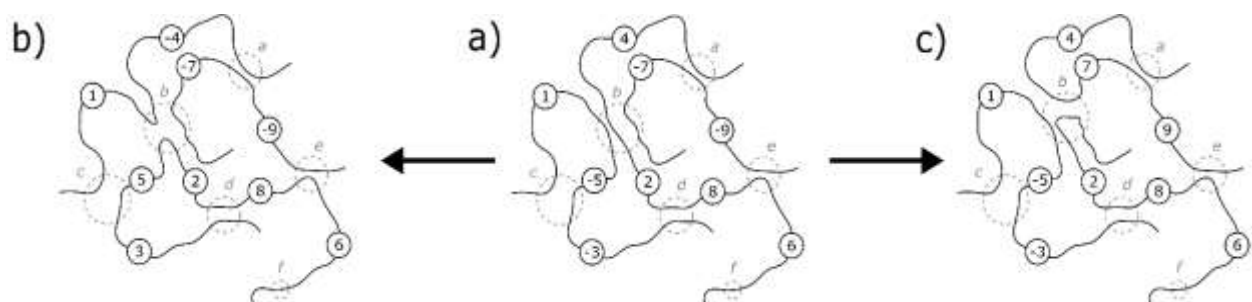


Fig 57. a) A 2D representation of a potential 3D configuration for the genome $G_A = \{(\circ 4, 2, 8, 6 \circ) (\circ 1, -5, -3 \circ) (\circ -9, -7 \circ)\}$. The numbers represent those genes making up a chromosome. The spatial proximity of adjacencies between genes are highlighted by the grey dashed circles. Groups of adjacencies in close proximity are labeled by the alphabetical letters $\{a, b, c, d, e, f\}$. b and c) Illustrations of two different ways in which the adjacencies located in group b could be broken and reformed.

Their paper catalyzed the publication of three follow up papers.

In [118] the validity of the method introduced in [117] is investigated and confirmed. It is shown that even a simple method of clustering adjacencies into groups based on Hi-C data, results in informative partitions, with the quality of the clustering directly correlated to the quality of the corresponding computed rearrangement scenario (rearrangement scenarios requiring fewer non-local rearrangements are considered to be of higher quality).

The authors also introduce an algorithm that bypasses the need for clustering adjacencies into groups by computing a DCJ scenario that greedily selects the next operation based on the values taken directly from the Hi-C contact map for the pair of breakpoints it involves.

In another paper [119] the parsimony criteria is disregarded, with the sole aim being the minimization of the number of ‘rare’ operations present in a sorting scenario. (A problem introduced in [117] as the *minimum local scenario (MLS)* problem). The authors show that the MLS problem is NP-complete and consequently introduce a 1.5-approximation algorithm addressing it. They also developed an exact algorithm for the problem which is exponential in the number of colors used to classify adjacencies but not in the number of genes present in the genomes.

A new method, based on the greedy algorithm presented in [118], extends the applicability to genomes of unequal gene content [120]. The concept of ‘ghost’ adjacencies introduced in [94] is used, enabling the algorithm to account for single gene insertions and deletions, as well as gene duplications.

5.3. Final Thoughts

The bioinformatics tool, Genolve, implements a biologically accurate genomic sorting model (an amended version of the rDCJ model) incorporated into an algorithm that finds all the most parsimonious ways in which one genome can be sorted into another.

This, to the best of my knowledge, is the first open-source tool able to generate not a single but all most parsimonious solutions. In addition, the amendment made to the original rDCJ model has led to increased accuracy in the model used for genomic sorting.

Genolve will assist researchers interested in conducting comprehensive studies of the process by which one genome evolves into another.

Reference List

1. Bakloushinskaya I. *Chromosomal rearrangements, genome reorganization, and speciation*. Biology Bulletin. 2016;43:759–775. doi:10.1134/S1062359016080057.
2. Eichler EE, Sankoff D. *Structural Dynamics of Eukaryotic Chromosome Evolution*. Science. 2003;301:793–797. doi: 10.1126/science.1086132.
3. Stratton MR, Campbell PJ, Futreal PA. *The cancer genome*. Nature. 2009;458:719–724. doi: 10.1038/nature07943.
4. Notta F, Chan-Seng-Yue M, Lemire M, et al. *A renewed model of pancreatic cancer evolution based on genomic rearrangement patterns*. Nature. 2016; 538:378–382. doi: 10.1038/nature19823.
5. Stankiewicz P, Lupski JR. *Genome architecture, rearrangements and genomic disorders*. Trends in Genetics. 2002;18:74–82. doi: 10.1016/s0168-9525(02)02592-1.
6. Carvalho CMB, Lupski JR. *Mechanisms underlying structural variant formation in genomic disorders*. Nature Reviews Genetics. 2016;17:224–238. doi: 10.1038/nrg.2015.25.
7. Huang CRL, Burns KH, Boeke JD. *Active Transposition in Genomes*. Annual Review of Genetics. 2012;46:651–675. doi: 10.1146/annurev-genet-110711-155616.
8. Fedoroff N. *Transposons and genome evolution in plants*. Proceedings of the National Academy of Sciences. 2000;97:7002–7007. doi: 10.1073/pnas.97.13.7002.
9. Albrecht-Buehler G. *Asymptotically increasing compliance of genomes with Chargaff's second parity rules through inversions and inverted transpositions*. Proceedings of the National Academy of Sciences. 2006; 103:17828–17833. doi: 10.1073/pnas.0605553103.
10. Miklos I. *MCMC genome rearrangement*. Bioinformatics. 2003;19:ii130–ii137. doi: 10.1093/bioinformatics/btg1070.
11. Sturtevant AH. *Genetic Factors Affecting the Strength of Linkage in Drosophila*. Proceedings of the National Academy of Sciences of the United States of America. 1917;3:555–558. doi: 10.1073/pnas.3.9.555.
12. Dunham MJ, Badrane H, Ferea T, et al. *Characteristic genome rearrangements in experimental evolution of Saccharomyces cerevisiae*. Proceedings of the National Academy of Sciences. 2002;99:16144–16149. doi: 10.1073/pnas.242624799.
13. Pevzner P. *Genome Rearrangements in Mammalian Evolution: Lessons From Human and Mouse Genomes*. Genome Research. 2003;13:37–45. doi: 10.1101/gr.757503.
14. Bailey JA, Baertsch R, Kent WJ, et al. *Hotspots of mammalian chromosomal evolution*. Genome Biology. 2004;5:R23. doi: 10.1186/gb-2004-5-4-r23.

15. Edger PP, Poorten TJ, VanBuren R, et al. *Origin and evolution of the octoploid strawberry genome*. Nature Genetics. 2019;51:541–547. doi: 10.1038/s41588-019-0356-4.
16. García-Ríos E, Nuévalos M, Barrio E, et al. *A new chromosomal rearrangement improves the adaptation of wine yeasts to sulfite*. Environmental Microbiology. 2019;21:1771–1781. doi: 10.1111/1462-2920.14586.
17. Sturtevant AH, Dobzhansky T. *Inversions in the Third Chromosome of Wild Races of Drosophila Pseudoobscura, and Their Use in the Study of the History of the Species*. Proceedings of the National Academy of Sciences. 1936;22:448–450. doi: 10.1073/pnas.22.7.448.
18. Dobzhansky T, Sturtevant AH. *Inversions in the Chromosomes of Drosophila Pseudoobscura*. Genetics. 1938;23:28–64. PMID: 17246876; PMCID: PMC1209001.
19. Sturtevant AH, Novitski E. *The Homologies of the Chromosome Elements in the Genus Drosophila*. Genetics. 1941;26:517–41. PMID: 17247021; PMCID: PMC1209144.
20. Hein J. *Reconstructing evolution of sequences subject to recombination using parsimony*. Mathematical Biosciences. 1990;98:185–200. doi: 10.1016/0025-5564(90)90123-G.
21. Fertin G, Labarre A, Rusu I, et al. *Combinatorics of Genome Rearrangements*. The MIT Press, Cambridge, MA. 2009.
22. Palmer JD, Herbon LA. *Plant mitochondrial DNA evolved rapidly in structure, but slowly in sequence*. Journal of Molecular Evolution. 1988;28:87–97. doi: 10.1007/BF02143500.
23. Watterson GA, Ewens WJ, Hall TE, et al. *The chromosome inversion problem*. Journal of Theoretical Biology. 1982;99:1–7. doi: 10.1016/0022-5193(82)90384-8.
24. Kececioğlu JD, Sankoff D. *Exact and approximation algorithms for the inversion distance between two chromosomes*. Combinatorial Pattern Matching - 4th Annual Symposium, CPM 1993, Proceedings. 1993;684 LNCS:87–105.
25. Zimao Li, Lusheng Wang, Kaizhong Zhang. *Algorithmic approaches for genome rearrangement: a review*. IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews). 2006;36:636–648. doi: 10.1109/TSMCC.2005.855522
26. Bafna V, Pevzner PA. *Genome rearrangements and sorting by reversals*. Annual Symposium on Foundations of Computer Science (Proceedings). 1993;148–157.
27. Hannenhalli S, Pevzner P. *Transforming cabbage into turnip*. Proceedings of the twenty-seventh annual ACM symposium on Theory of computing - STOC '95. 1995;178–189. doi: 10.1145/225058.225112.
28. Kaplan H, Shamir R, Tarjan RE. *A Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals*. SIAM Journal on Computing. 2000;29:880–892. doi: 10.1137/S0097539798334207.

29. Bader DA, Moret BME, Yan M. *A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study*. Journal of Computational Biology. 2001;8:483–491. doi: 10.1089/106652701753216503.
30. Bergeron A. *A Very Elementary Presentation of the Hannenhalli-Pevzner Theory*. Combinatorial Pattern Matching - 4th Annual Symposium, CPM 2001, Proceedings. 2001;106–117. doi: 10.5555/1704846.1705128.
31. Bergeron A, Mixtacki J, Stoye J. *Reversal Distance without Hurdles and Fortresses*. Combinatorial Pattern Matching - 4th Annual Symposium, CPM 2004, Proceedings. 2004;388–399.
32. Tannier E, Sagot M-F. *Sorting by Reversals in Subquadratic Time*. Combinatorial Pattern Matching - Annual Symposium, CPM 2004, Proceedings. 2004. doi: 10.1007/978-3-540-27801-6_1
33. Tannier E, Bergeron A, Sagot M-F. *Advances on sorting by reversals*. Discrete Applied Mathematics. 2007;155:881–888. doi: 10.1016/j.dam.2005.02.033
34. Seoighe C, Federspiel N, Jones T, et al. *Prevalence of small inversions in yeast gene order evolution*. Proceedings of the National Academy of Sciences. 2000;97:14433–14437. doi: 10.1073/pnas.240462997
35. Hannenhalli S. *Polynomial-time algorithm for computing translocation distance between genomes*. Discrete Applied Mathematics. 1996;71:137–151. doi: 10.1016/S0166-218X(96)00061-3
36. Bergeron A, Mixtacki J, Stoye J. *On Sorting by Translocations*. Journal of Computational Biology. 2006;13:567–578. doi: 10.1089/cmb.2006.13.567.
37. Ozery-Flato M, Shamir R. *Sorting by Translocations Via Reversals Theory*. Journal of Computational Biology. 2007;14:4. doi: 10.1089/cmb.2007.A003.
38. Hannenhalli S, Pevzner PA. *Transforming men into mice (polynomial algorithm for genomic distance problem)*. Proceedings of IEEE 36th Annual Foundations of Computer Science. 1995;581–592. doi: 10.1.1.50.5495.
39. Tesler G. *Efficient algorithms for multichromosomal genome rearrangements*. Journal of Computer and System Sciences. 2002;65:587–609. doi: 10.1016/S0022-0000(02)00011-9.
40. Ozery-Flato M. *Two notes on genome rearrangements*. Journal of Bioinformatics and Computational Biology. 2003;01:71–94. doi: 10.1142/S0219720003000198.
41. Jean G, Nikolski M. *Genome rearrangements: a correct algorithm for optimal capping*. Information Processing Letters. 2007;104:14–20. doi: 10.1016/j.ipl.2007.04.011.

42. Bulteau L, Fertin G, Rusu I. *Sorting by Transpositions Is Difficult*. In: Aceto L., Henzinger M., Sgall J. (eds) Automata, Languages and Programming. ICALP 2011. Lecture Notes in Computer Science, vol 6755: 654–665. Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-22006-7_55.
43. Christie DA. *A 3/2-approximation Algorithm for Sorting by Reversals*. Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. 1998;244–252. doi: 10.5555/314613.314711.
44. Bafna V, Pevzner P. *Sorting Permutations by Transpositions*. Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms. 1995;614–623. doi: 10.5555/313651.313826.
45. Hartman T. *A Simpler 1.5-Approximation Algorithm for Sorting by Transpositions*. Combinatorial Pattern Matching. 2003;156–169. doi: 10.1016/j.jcss.2004.12.006.
46. Hartman T, Shamir R. *A simpler and faster 1.5-approximation algorithm for sorting by transpositions*. Information and Computation. 2006;204:275–290. doi: 10.1016/j.ic.2005.09.002.
47. Elias I, Hartman T. *A 1.375-Approximation Algorithm for Sorting by Transpositions*. IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2006;3:369–379. doi: 10.1109/TCBB.2006.44.
48. Gu Q-P, Peng S, Sudborough H. *A 2-approximation algorithm for genome rearrangements by reversals and transpositions*. Theoretical Computer Science. 1999;210:327–339. doi: 10.1016/S0304-3975(98)00092-9.
49. Eriksen N. *(1+ε)-Approximation of sorting by reversals and transpositions*. Theoretical Computer Science. 2002;289:517–529. doi: 10.1016/S0304-3975(01)00338-3.
50. Rahman A, Shatabda S, Hasan M. *An Approximation Algorithm for Sorting by Reversals and Transpositions*. Journal of Discrete Algorithms. 2008;6:449–457. doi: 10.1016/j.jda.2007.09.002.
51. Christie D, Irving R. *Sorting Strings by Reversals and by Transpositions*. SIAM Journal of Discrete Mathematics. 2001;14:193–206. doi: 10.1137/S0895480197331995.
52. Christie DA. *Sorting permutations by block-interchanges*. Information Processing Letters. 1996;60:165–169. doi: 10.1016/S0020-0190(96)00155-X.
53. Feng J, Zhu D. *Faster algorithms for sorting by transpositions and sorting by block-interchanges*. In: TAMC 2006, Beijing, China, 2006. Pg. 128–137. doi: 10.1145/1273340.1273341.
54. Hartmann T, Middendorf M, Bernt M. *Genome Rearrangement Analysis: Cut and Join Genome Rearrangements and Gene Cluster Preserving Approaches*. Methods Molecular Biology. 2018;1704:261–289. doi: 10.1007/978-1-4939-7463-4_9.
55. Bergeron A, Mixtacki J, Stoye J. *A Unifying View of Genome Rearrangements* In WABI '06 proceedings of the sixth international workshop on algorithms in bioinformatics. 2006 Vol. 4175 of LNBI. pp. 163–173. doi: 10.1007/11851561_16.

56. Feijão P, Meidanis J. *SCJ :A Variant of Breakpoint Distance for Which Sorting Genome Median and Genome Halving Problems Are Easy*. WABI '09: Proc. Ninth Int'l Conf. Algorithms in Bioinformatics, pp. 85-96, 2009. doi: 10.1007/978-3-642-04241.
57. Feijão P, Meidanis J. *SCJ: A Breakpoint-Like Distance that Simplifies Several Rearrangement Problems*. IEEE/ACM transactions on computational biology and bioinformatics. 2011;8:1318–1329. doi: 10.1109/TCBB.2011.34.
58. Medvedev P, Stoye J. *Rearrangement Models and Single-Cut Operations*. Comparative Genomics. 2009;84–97. doi: 10.1007/978-3-642-04744-2_8.
59. Yancopoulos S, Attie O, Friedberg R. *Efficient sorting of genomic permutations by translocation, inversion and block interchange*. Bioinformatics. 2005;21:3340–3346. doi: 10.1093/bioinformatics/bti535.
60. Hilker R, Sickinger C, Pedersen CNS, et al. *UniMoG—a unifying framework for genomic distance calculation and sorting based on DCJ*. Bioinformatics. 2012;28:2509–2511. doi: 10.1093/bioinformatics/bts440.
61. Kováč J, Warren R, Braga MD V, et al. *Restricted DCJ Model: Rearrangement Problems with Chromosome Reincorporation*. Journal of Computational Biology. 2011;18:1231–1241. doi: 10.1089/cmb.2011.0116 .
62. Darling AC, Mau B, Blattner FR, Perna NT. *Mauve: multiple alignment of conserved genomic sequence with rearrangements*. Genome Research. 2004;14(7):1394-403. doi: 10.1101/gr.2289704.
63. Tesler G. *GRIMM: genome rearrangements web server*. Bioinformatics. 2002;18(3):492-493. doi: 10.1093/bioinformatics/18.3.492.
64. Braga MDV, Stoye J. *The Solution Space of Sorting by DCJ*. Journal of Computational Biology. 2010;17:1145–1165. doi: 10.1089/cmb.2010.0109.
65. Braga MDV, Sagot M-F, Scornavacca C, et al. *Exploring the Solution Space of Sorting by Reversals, with Experiments and an Application to Evolution*. IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2008;5:348–356. doi: 10.1109/TCBB.2008.16.
66. Sung W-K. *Algorithms in Bioinformatics: A Practical Introduction*. CRC Press (Taylor & Francis Group), 1st edition, 2010.
67. Kováč J, Warren R, Braga MDV, et al. *Restricted DCJ Model: Rearrangement Problems with Chromosome Reincorporation*. Journal of Computational Biology. 2011;18:1231–1241. doi: 10.1089/cmb.2011.0116.
68. Bailey JA, Baertsch R, Kent WJ, et al. *Hotspots of mammalian chromosomal evolution*. Genome Biology. 2004;5:R24. doi: 10.1186/gb-2004-5-4-r23.

69. Liti G. *The fascinating and secret wild life of the budding yeast S. cerevisiae*. Elife. 2015;4:e05835. doi:10.7554/eLife.05835.
70. McGovern PE, Zhang J, Tang J, et al. *Fermented beverages of pre- and proto-historic China*. PNAS. 2004;101:17593–17598. doi: 10.1073/pnas.0407921102.
71. Duina AA, Miller ME, Keeney JB. *Budding Yeast for Budding Geneticists: A Primer on the Saccharomyces cerevisiae Model System*. Genetics. 2014;.197:33–48. doi: 10.1534/genetics
72. Goffeau A, Barrell BG, Bussey H, et al. *Life with 6000 genes*. Science. 1996;274:546, 563–567. doi: 10.1126/science.274.5287.546..
73. Murakami C, Kaeberlein M. *Quantifying yeast chronological life span by outgrowth of aged cells*. J Vis Exp. 2009;6(27):1156. doi: 10.3791/1156.
74. Cussiol JRR, Soares BL, Oliveira FMB de, et al. *From yeast to humans: Understanding the biology of DNA Damage Response (DDR) kinases*. Genetics and Molecular Biology. 2020;43. doi: 10.1590/1678-4685-gmb-2019-0071 .
75. Biddick R, Young ET. *The disorderly study of ordered recruitment*. Yeast. 2009;26:205–220. doi: 10.1002/yea.1660.
76. Feyder S, De Craene J-O, Bär S, et al. *Membrane Trafficking in the Yeast Saccharomyces cerevisiae Model*. International Journal of Molecular Sciences. 2015;16:1509–1525. doi: 10.3390/ijms16011509.
77. Altmann K, Dürr M, Westermann B. *Saccharomyces cerevisiae as a Model Organism to Study Mitochondrial Biology*. Mitochondria: Practical Protocols. 2007;81–90. doi: 10.1007/978-1-59745-365-3_6.
78. Hohmann S, Krantz M, Nordlander B. *Yeast osmoregulation*. Methods Enzymol. 2007;428:29–45. doi: 10.1016/S0076-6879(07)28002-4.
79. Nasheuer H-P, Smith R, Bauerschmidt C, et al. *Initiation of eukaryotic DNA replication: regulation and mechanisms*. Prog Nucleic Acid Research Molecular Biology. 2002;72:41–94. doi: 10.1016/s0079-6603(02)72067-9.
80. Owsianowski E, Walter D, Fahrenkrog B. *Negative regulation of apoptosis in yeast*. Biochim Biophys Acta. 2008;1783:1303–1310. doi: 10.1016/j.bbamcr.2008.03.006.
81. Duan S-F, Han P-J, Wang Q-M, et al. *The origin and adaptive evolution of domesticated populations of yeast from Far East Asia*. Nature Communications. 2018;9:2690. doi: 10.1038/s41467-018-05106-7.
82. Wang Q-M, Liu W-Q, Liti G, et al. *Surprisingly diverged populations of Saccharomyces cerevisiae in natural environments remote from human activity*. Molecular Ecology. 2012;21:5404–5417. doi: 10.1111/j.1365-294X.2012.05732.x.

83. Naumov GI, Gazdiev DO, Naumova ES. *The Finding of the Yeast Species Saccharomyces bayanus in Far East Asia*. Microbiology. 2003;72:738–743. doi: 10.1023.
84. J Bing, P Han, W Liu, et al. *Evidence for a Far East Asian origin of lager beer yeast*. Current biology. 2014;24(10):R380-1. doi: 10.1016/j.cub.2014.04.031.
85. V Gayevskiy, MR Goddard. *Saccharomyces eubayanus and Saccharomyces arboricola reside in North Island native New Zealand forests*. Environmental microbiology. 2016;18(4):1137-47. doi: 10.1111/1462-2920.13107.
86. Darling AE, Treangen TJ, Messeguer X, et al. *Analyzing Patterns of Microbial Evolution Using the Mauve Genome Alignment System*. Methods in Molecular Biology. 2007;396:135–152. doi: 10.1007/978-1-59745-515-2_10.
87. Zhang Z, Schwartz S, Wagner L, Miller W. *A greedy algorithm for aligning DNA sequences*. Journal of Computational Biology. 2000;7(1-2):203-14. doi: 10.1089/10665270050081478.
88. Guo Q, Koirala S, Perkins EM, McCaffery JM, Shaw JM. *The Mitochondrial Fission Adaptors Caf4 and Mdv1 Are Not Functionally Equivalent*. PLoS ONE. 2012;7(12):e53523. doi: 10.1371/journal.pone.0053523.
89. Pairwise Sequence Alignment Tools < EMBL-EBI: <https://www.ebi.ac.uk/Tools/psa/>
90. Sankoff D. *Genome rearrangement with gene families*. Bioinformatics. 1999;15(11):909-17. doi: 10.1093/bioinformatics/15.11.909.
91. Angibaud S, Fertin G, Rusu I, et al. *Efficient tools for computing the number of breakpoints and the number of adjacencies between two genomes with duplicate genes*. Journal of computational biology : a journal of computational molecular cell biology. 2008;15(8):1093-1115. doi: 10.1089/cmb.2008.0061.
92. Blin G, Fertin G, Chauve C. *The breakpoint distance for signed sequences*. 1st Conference on Algorithms and Computational Methods for biochemical and Evolutionary Networks (CompBioNets'04) 2004.
93. Zeira R, Shamir R. *Genome Rearrangement Problems with Single and Multiple Gene Copies: A Review*. In: Bioinformatics and Phylogenetics. Springer: 2019. p. 205–41. doi: 10.1007/978-3-030-10837-3_10.
94. Yancopoulos S, Friedberg R. *Sorting Genomes with Insertions, Deletions and Duplications by DCJ*. Comparative Genomics. RECOMB-CG 2008. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. 2008;5267:170–183. doi: 10.1007/978-3-540-87989-3_13.
95. Braga MD V, Willing E, Stoye J. *Double Cut and Join with Insertions and Deletions*. Journal of Computational Biology. 2011;18(9):1167–1184. doi: 10.1089/cmb.2011.0118.

96. da Silva PH, Machado R, Dantas S, et al. *Restricted DCJ-indel model: sorting linear genomes with DCJ and indels*. BMC bioinformatics. 2012;13 Suppl 19(Suppl 19):S14. doi: 10.1186/1471-2105-13-S19-S14.
97. Braga MD V, Stoye J. *Sorting Linear Genomes with Rearrangements and Indels*. IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2015;12:500–506. doi: 10.1109/TCBB.2014.2329297.
98. Compeau PEC. *A Simplified View of DCJ-Indel Distance*. In: Raphael B., Tang J. (eds) Algorithms in Bioinformatics. WABI 2012. Lecture Notes in Computer Science, vol 7534. Springer, Berlin, Heidelberg. pg 365–377. doi: 10.1007/978-3-642-33122-0_29.
99. Compeau PE. *DCJ-Indel sorting revisited*. Algorithms Molecular Biology. 2013;8(1):6. doi:10.1186/1748-7188-8-6.
100. Bader M. *Sorting by reversals, block interchanges, tandem duplications, and deletions*. BMC bioinformatics. 2009;10:S9. doi: 10.1186/1471-2105-10-S1-S9.
101. Bader M. *Genome rearrangements with duplications*. BMC bioinformatics. 2010;11:S27. doi: 10.1186/1471-2105-11-S1-S27.
102. Shao M, Lin Y, Moret B. *Sorting genomes with rearrangements and segmental duplications through trajectory graphs*. BMC bioinformatics 2013; 14:S9. doi: 10.1186/1471-2105-14-S15-S9.
103. Paten B, Zerbino DR, Hickey G, et al. *A unifying model of genome evolution under parsimony*. BMC bioinformatics. 2014;15:206. doi: 10.1186/1471-2105-15-206.
104. Darling AE, Miklos I, Ragan MA. *Dynamics of genome rearrangement in bacterial populations*. PLoS Genetics. 2008;18:4(7):e1000128. doi: 10.1371/journal.pgen.1000128.
105. Lefebvre JF, El-Mabrouk N, Tillier E, et al. *Detection and validation of single gene inversions*. Bioinformatics. 2003;19 Suppl 1:i190-6. doi: 10.1093/bioinformatics/btg1025.
106. Sankoff D, Lefebvre JF, Tillier E, et al. *The distribution of inversion lengths in bacteria*. Lecture Notes in Bioinformatics (Subseries of Lecture Notes in Computer Science). 2005;3388:97–108.
107. Dias Z, Dias U, Heath LS, et al. *Sorting genomes using almost-symmetric inversions*. Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12. 2012;1368. doi: 10.1145/2245276.2231993.
108. Baudet C, Dias U, Dias Z. *Sorting by weighted inversions considering length and symmetry*. BMC bioinformatics. 2015;16 Suppl 19:S3. doi: 10.1186/1471-2105-16-S19-S3.

109. Nadeau JH, Taylor BA. *Lengths of chromosomal segments conserved since divergence of man and mouse*. Proceedings of the National Academy of Sciences of the United States of America. 1984;81:814–818. doi: 10.1073/pnas.81.3.814.
110. Biller P, Gueguen L, Knibbe C, et al. *Breaking Good: Accounting for Fragility of Genomic Regions in Rearrangement Distance Estimation*. Genome biology and evolution. 2016;8(5):1427–1439. doi: 10.1093/gbe/evw083.
111. Fertin G, Jean G, Tannier E. *Algorithms for computing the double cut and join distance on both gene order and intergenic sizes*. Algorithms for Molecular Biology. 2017;12:16. doi: 10.1186/s13015-017-0107-y.
112. Bulteau L, Fertin G, Tannier E. *Genome rearrangements with indels in intergenes restrict the scenario space*. BMC bioinformatics. 2016;17:426. doi: 10.1186/s12859-016-1264-6.
113. Véron AS, Lemaitre C, Gautier C, et al. *Close 3D proximity of evolutionary breakpoints argues for the notion of spatial synteny*. BMC Genomics. 2011;12:303. doi: 10.1186/1471-2164-12-303.
114. Swenson K, Blanchette M. *Large-scale mammalian genome rearrangements coincide with chromatin interactions*. Bioinformatics. 2019;15:35(14):i117-i126. doi: 10.1093.
115. Roix JJ, McQueen PG, Munson PJ, et al. *Spatial proximity of translocation-prone gene loci in human lymphomas*. Nature Genetics. 2003;34(3):287–291. doi: 10.1038/ng1177.
116. Mani R-S, Chinnaiyan A. *Triggers for genomic rearrangements: Insights into genomic*. Nature Reviews Genetics. 2010;11(12):819–829. doi: 10.1038/nrg2883.
117. Swenson KM, Simonaitis P, Blanchette M. *Models and algorithms for genome rearrangement with positional constraints*. Algorithms for molecular biology. 2016;11:13. doi: 10.1186/s13015-016-0065-9.
118. Pulicani S, Simonaitis P, Rivals E, et al. *Rearrangement Scenarios Guided by Chromatin Structure*. Comparative Genomics. 2017;141–155. doi: 10.1007/978-3-319-67979-2_8.
119. Simonaitis P, Swenson KM. *Finding local genome rearrangements*. Algorithms for Molecular Biology. 2018;13:9. doi: 10.1186/s13015-018-0127-2.
120. Simonaitis P, Chateau A, Swenson KM. *A general framework for genome rearrangement with biological constraints*. Algorithms for molecular biology. 2019;14:15. doi: 10.1186/s13015-019-0149-4.